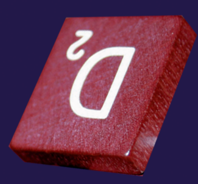
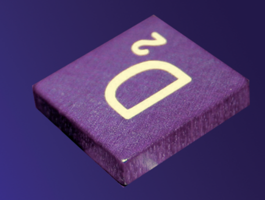
Estructuras de datos en Java



Luis Joyanes Aguilar
Ignacio Zahonero Martínez









Estructuras de datos en Java



Estructuras de datos en Java

Luis Joyanes Aguilar

Ignacio Zahonero Martínez



MADRID • BOGOTÁ • BUENOS AIRES • CARACAS • GUATEMALA • LISBOA • MÉXICO
NUEVA YORK • PANAMÁ • SANTIAGO • SÃO PAULO
AUCKLAND • HAMBURGO • LONDRES • MILÁN • MONTREAL • NUEVA DELHI • PARÍS
SAN FRANCISCO • SIDNEY • SINGAPUR • ST. LOUIS • TOKIO • TORONTO

ESTRUCTURAS DE DATOS EN JAVA.

Edificio Valrealty, 1. ª Planta
Basauri, 17
28023 Aravaca (Madrid)

ISBN: 978-84-481-5631-2
Depósito legal:

Editor: José Luis García
Técnico Editorial: Blanca Pecharromán
Compuesto en: Gesbiblo, S. L.
Diseño de cubierta: Gesbiblo, S. L.
Impreso por:

IMPRESO EN ESPAÑA – PRINTED IN SPAIN



A Luis y Paloma.



PRÓLOGO

Dos de las disciplinas clásicas en todas las carreras relacionadas con la Informática y las Ciencias de la Computación son *Estructuras de Datos* y *Algoritmos,* o bien una sola disciplina si ambas se estudian integradas: *"Algoritmos y Estructuras de Datos".* El estudio de estructuras de datos y de algoritmos es tan antiguo como el nacimiento de la programación y se ha convertido en estudio obligatorio en todos los currículos desde finales de los años 70 y sobre todo en esa misma década cuando apareció el lenguaje Pascal de la mano del profesor Niklaus Wirtz, y posteriormente en la década de los ochenta con la aparición de su obra —ya clásica— *Algorithms and Data Structures* (1986).

Muchas facultades y escuelas de Ingeniería, así como institutos tecnológicos, comienzan sus cursos de Estructuras de Datos con el soporte de Java. Existen muchas razones por las cuales pensamos que Java es apropiado para la formación en estructuras de datos. Una de ellas es que Java, es un lenguaje más moderno que C o C++, con mejores funcionalidades, orientado a objetos, a la programación en Web,… Además, a partir de Java 1.5 permite diseñar clases genéricas, de forma similar a las plantillas *(templates)* de C++.

El primer problema que se suele presentar al estudiante de *Estructura de Datos* que, probablemente, procederá de un curso de nivel básico, medio o avanzado de *introducción o fundamentos de programación* o bien de *iniciación en algoritmos*, es precisamente el modo de afrontar información compleja desde el principio. Aunque es verdad que Java[1] tiene muchas ventajas sobre un lenguaje procedimental, por ejemplo C, muchas de estas ventajas no se hacen evidentes hasta que un programa se "vuelve" o "hace" más complejo. En este caso el *paradigma orientado a objetos* es una herramienta de programación y organización muy poderosa y con grandes ventajas para la enseñanza y posterior tarea profesional.

A primera vista, Java es más interesante que un lenguaje procedimental por su enfoque orientado a objetos, aunque puede parecer, en el caso del análisis y diseño de algoritmos y estructuras de datos, que esta propiedad añade una complejidad inherente y no es así, la implementación en clases y objetos puede darle una nueva potencialidad. Pensando en esta transición se ha incluido un capítulo dedicado a conceptos teórico-prácticos de orientación a objetos. En cualquier caso, el curso está soportando la comprensión del tipo abstracto de datos (TAD) de modo que el estilo de programación empleado en el texto se basa en el estudio de tipos abstractos de datos como base para la formación en orientación a objetos.

Se estudian estructuras de datos con un objetivo fundamental: aprender a escribir programas más eficientes. También cabe aquí hacerse la pregunta: ¿Por qué se necesitan programas más eficientes cuando las nuevas computadoras son más rápidas cada año? La razón tal vez resida en el hecho de que nuestras metas no se amplían a medida que se aumentan las características de las computadoras. La potencia de cálculo y las capacidades de almacenamiento aumentan la eficacia y ello conlleva un aumento de los resultados de las máquinas y de los programas desarrollados por ellas.

---

[1] Véanse otras obras de los autores, publicadas también en McGraw-Hill, tales como *Programación en C++, Programación en Java 2* o *Programación en C.*

La búsqueda de la eficiencia de un programa no debe chocar con un buen diseño y una codificación clara y legible. La creación de programas eficientes tiene poco que ver con "trucos de programación" sino, al contrario, se basan en una buena organización de la información y buenos algoritmos. Un programador que no domine los principios básicos de diseños claros y limpios probablemente no escribirá programas eficientes. A la inversa, programas claros requieren organizaciones de datos claras y algoritmos claros, precisos y transparentes.

La mayoría de los departamentos informáticos reconocen que las destrezas de buena programación requieren un fuerte énfasis en los principios básicos de ingeniería de software. Por consiguiente, una vez que un programador ha aprendido los principios para diseñar e implementar programas claros y precisos, el paso siguiente es estudiar los efectos de las organizaciones de datos y los algoritmos en la eficiencia de un programa.

EL ENFOQUE DEL LIBRO

En esta obra se muestran numerosas técnicas de representación de datos. En su contexto las mismas se engloban en los siguientes principios:

1. Cada estructura de datos tiene sus costes y sus beneficios. Los programadores y diseñadores necesitan una comprensión rigurosa y completa de cómo evaluar los costes y beneficios para adaptarse a los nuevos retos que afronta la construcción de la aplicación. Estas propiedades requieren un conocimiento o comprensión de los principios del análisis de algoritmos y también una consideración práctica de los efectos significativos del medio físico empleado (p.e. datos almacenados en un disco frente a memoria principal).
2. Los temas relativos a costes y beneficios se consideran dentro del concepto de elemento de compensación. Por ejemplo, es bastante frecuente reducir los requisitos de tiempo en beneficio de un incremento de requisitos de espacio en memoria, o viceversa.
3. Los programadores no deben reinventar la rueda continuamente. Por consiguiente, los estudiantes necesitan aprender las estructuras de datos utilizadas junto con los algoritmos correspondientes.
4. Los datos estructurados siguen a las necesidades. Los estudiantes deben aprender a evaluar primero las necesidades de la aplicación y, a continuación, encontrar una estructura de datos en correspondencia con sus funcionalidades.

Esta edición describe, fundamentalmente, *estructuras de datos,* métodos de organización de grandes cantidades de datos y *algoritmos* junto con el *análisis de los mismos,* en esencia estimación del tiempo de ejecución de algoritmos. A medida que las computadoras se vuelven más y más rápidas, la necesidad de programas que pueden manejar grandes cantidades de entradas se vuelve más críticas y su eficiencia aumenta a medida que estos programas pueden manipular más y mejores organizaciones de datos. Analizando un algoritmo antes de que se codifique realmente, los estudiantes pueden decidir si una determinada solución será factible y rigurosa. Por ejemplo, se pueden ver cómo diseños e implementaciones cuidadas pueden reducir los costes en tiempo y memoria. Por esta razón, se dedican dos capítulos, en exclusiva a tratar los conceptos fundamentales de *análisis de algoritmos*, y en un gran número de algoritmos se incluyen explicaciones de tiempos de ejecución para poder medir la complejidad y eficiencia de los mismos.

El método didáctico que sigue nuestro libro ya lo hemos seguido en otras obras nuestras y busca preferentemente enseñar al lector a pensar en la resolución de un problema siguiendo un determinado método ya conocido o bien creado por el propio lector. Una vez esbozado el método, se estudia el algoritmo correspondiente junto con las etapas que pueden resolver el problema. A continuación, se escribe el algoritmo, en la mayoría de las veces en lenguaje Java. Para que el lector pueda verificar su programa en la computadora, se incluyen los códigos fuente en la página web del libro.

Uno de los objetivos fundamentales es enseñar al estudiante, simultáneamente, buenas reglas de programación y análisis de algoritmos de modo que puedan desarrollar los programas con la mayor eficiencia posible.

Aunque se ha tratado de que el material de este texto sea lo menos dependiente del lenguaje, la programación, como el lector sabe, requiere el uso de un lenguaje específico. En nuestro caso se ha elegido Java como espina dorsal de programación

Java ya es un lenguaje de programación muy extendido en el mundo de la programación y de la ingeniería de software, tal vez el más utilizado por su gran vinculación con Internet y la web, y también en comparación con el otro gran lenguaje de programación C++. Java ofrece muchos beneficios y los programadores suelen verlo como más seguro, más portable y más fácil de utilizar que C++. Por estas propiedades, es un lenguaje idóneo para el examen e implementación de estructuras de datos fundamentales. Otras características importantes de Java, tales como hilos *(threads)* y sus *GUIs* (Interfaces gráficas de usuario), aunque importantes, no se suelen necesitar en este texto y, por consiguiente, no se han examinado.

Java es un lenguaje de programación sencillo, orientado a objetos, distribuido, interpretado, robusto, seguro, neutro ante la arquitectura, portable, altas prestaciones, multihilo y dinámico

Por el contrario, Java tiene pocas pero algunas desventajas comparado con C++: no soporta bien programación genérica, características de E/S menos potentes, … Pensando en aquellos lectores que deseen conocer las características del lenguaje C++, en la página oficial del libro, el lector, el profesor y el maestro, podrán encontrar amplia información de C++ y también en obras nuestras, “hermanas” de este texto, tales como *Programación en Java* y *Estructuras de datos en C++,* que se encuentran en dicha página.

Los programas han sido compilados en diversos entornos tales como *JCreator y NetBeans* utilizando la plataforma J2SE5 conocida popularmente como versión 5. La reciente aparición de Java SE 6, también conocida como Java 6 puede ser utilizada también para el desarrollo de programas. Para consultar las nuevas características de la versión 6 de Java, presentada a finales del 2006, visite la página de Sun:

```
//java.sun.com/javase/6
```

Si desea descargarse alguna de las versiones de esta nueva versión de Sun acceda a la siguiente dirección web:

```
//java.sun.com/javase/downloads/index.jsp
```

EL LIBRO COMO TEXTO DE REFERENCIA UNIVERSITARIA Y PROFESIONAL

El estudio de *Algoritmos* y de *Estructuras de Datos* son disciplinas académicas que se incorporan a todos los planes de estudios universitarios de Ingeniería e Ingeniería Técnica en Informática, Ingeniería de Sistemas Computacionales y Licenciatura en Informática, así como a los planes de estudio de Formación Profesional e institutos politécnicos. Suele considerarse también a estas disciplinas como ampliaciones de las asignaturas de *Programación,* en cualquiera de sus niveles.

En el caso de España, los actuales planes de estudios de Ingeniería Técnica en Informática e Ingeniería Informática, y los futuros, contemplados en la Declaración de Bolonia, tienen materias troncales relativas tanto a *Algoritmos* como a *Estructuras de Datos.* Igual sucede en los países iberoamericanos donde también es común incluir estas disciplinas en los currículos de carreras de Ingeniería de Sistemas y Licenciaturas en Informática. ACM, la organización profesional norteamericana más prestigiosa a nivel mundial, incluye en las recomendaciones de sus diferentes currículos de carreras relacionadas con informática el estudio de materias de algoritmos y estructuras de datos. En el conocido *Computing Curricula* de 1992 se incluyen descriptores recomendados de *Programación y Estructura de Datos*, y en los últimos currículos publicados, *Computing Curricula* 2001 y 2005, se incluyen en las áreas **PF** de *Fundamentos de Programación (Programming Fundamentals, PF1 a PF4)* y **AL** de *Algoritmos y Complejidad (Algorithms and Complexity, AL1 a AL3).* En este libro se han incluido los descriptores más importantes tales como *Algoritmos y Resolución de Problemas, Estructuras de datos fundamentales, Recursión, Análisis de algoritmos básicos y estrategias de algoritmos.* Además se incluye un estudio de algoritmos de estructuras discretas tan importantes como *Árboles y Grafos.* Por último, se desarrolla y se ponen ejemplos de todas las colecciones presentes en Java.

Organización del libro

Esta obra está concebida como un libro didáctico y eminentemente práctico. Se pretende enseñar los principios básicos requeridos para seleccionar o diseñar las estructuras de datos que ayudarán a resolver mejor los problemas y a no a memorizar una gran cantidad de implementaciones. Por esta razón, se presentan numerosos ejercicios y problemas resueltos en su totalidad, siempre organizados sobre la base del análisis del problema y el algoritmo correspondiente en Java. Los lectores deben tener conocimientos a nivel de iniciación o nivel medio en programación. Es deseable haber cursado al menos un curso de un semestre de introducción a los algoritmos y a la programación, con ayuda de alguna herramienta de programación, preferentemente, y se obtendrá el mayor rendimiento si además se tiene conocimiento de un lenguaje estructurado como C.

El libro busca de modo prioritario enseñar al lector técnicas de programación de algoritmos y estructuras de datos. Se pretende aprender a programar practicando el análisis de los problemas y su codificación en Java.

Está pensado para un curso completo anual o bien dos semestres, para ser estudiado de modo independiente —por esta razón se incluyen las explicaciones y conceptos básicos de la teoría de algoritmos y estructuras de datos— o bien de modo complementario, exclusivamente como apoyo de libros de teoría o simplemente del curso impartido por el maestro o profesor en su aula de clase. Pensando en su uso totalmente práctico se ha optado por seguir una estructura similar

al libro *Algoritmos y Estructura de Datos* publicado por McGraw-Hill por los profesores Joyanes y Zahonero de modo que incluye muchos de los problemas y ejercicios propuestos en esta obra. En caso de realizar su estudio conjunto, uno actuaría como libro de texto, fundamentalmente, y el otro como libro de prácticas para el laboratorio y el estudio en casa o en un curso profesional.

Contenido

El contenido del libro sigue los programas clásicos de las disciplinas *Estructura de Datos* y/o *Estructuras de Datos y de la Información* respetando las directrices emanadas de los *Currícula* de 1991 y las actualizadas del 2001 y 2005 de ACM/IEEE, así como de los planes de estudio de Ingeniero Informático e Ingeniero Técnico en Informática de España y los de Ingeniero de Sistemas y Licenciado en Informática de muchas universidades latinoamericanas.

La Parte I, *Análisis de algoritmos y estructuras de datos básicas* describe el importante concepto de análisis de un algoritmo y las diferentes formas de medir su complejidad y eficiencia; asimismo se describen las estructuras de datos más simples tales como arrays, cadenas o estructuras. La Parte II, *Diseño de algoritmos (Recursividad, ordenación y búsqueda)* examina los algoritmos más utilizados en la construcción de cualquier programa tales como los relativos a búsqueda y ordenación, así como las potentes técnicas de manipulación de la recursividad. La Parte III, *Estructuras de datos lineales (Abstracción de datos, listas, pilas, colas y tablas hash)* constituye una de las partes avanzadas del libro y suele formar parte de cursos de nivel medio/alto en organización de datos. Por último, la Parte IV, *Estructuras de datos no lineales (Árboles, grafos y sus algoritmos)* constituye también una de las partes avanzadas del libro; su conocimiento y manipulación permitirán al programador obtener el máximo aprovechamiento en el diseño y construcción de sus programas. La descripción más detallada de los capítulos correspondientes se reseñan a continuación.

Capítulo 1. Algoritmos y estructuras de datos. Los tipos de datos y la necesidad de su organización en estructuras de datos es la parte central de este capítulo. El estudio de los conceptos de algoritmos y programas y su herramienta de representación más característica, el *pseudocódigo,* son uno de los objetivos más ambiciosos de esta obra. El capítulo hace una revisión de sus propiedades más importantes: representación, eficiencia y exactitud. Se describe la notación *O grande* utilizada preferentemente en el análisis de algoritmos.

Capítulo 2. Tipos de datos: Clases y objetos. La programación orientada a objetos es, hoy en día, el eje fundamental de la programación de computadoras. Su núcleo esencial son los conceptos de clases y objetos. En el capítulo se consideran los conceptos teóricos de encapsulación de datos y tipos abstractos de datos como soporte de una clase y de un objeto; también se analiza el modo de construcción de objetos, así como conceptos tan importantes como la visibilidad de los miembros de una clase. Constructores, paquetes y recolección de objetos junto con la definición de tipos abstractos de datos en Java completan el capítulo.

Capítulo 3. Arrays (arreglos) y cadenas. Los diferentes tipos de *arrays (arreglos)* se describen y detallan junto con la introducción a las cadenas *(strings)*. La importante clase `String` se describe con detalle, así como la especificación de la clase `Vector`.

Capítulo 4. Clases derivadas y polimorfismo. Uno de los conceptos más empleados en programación orientada a objetos y que ayudará al programador de un modo eficiente al diseño de estructura de datos son las *clases derivadas.* La propiedad de *herencia*, junto con el *polimorfismo* ayudan a definir con toda eficacia las clases derivadas. Otro término fundamental

en POO son las *clases abstractas* que permitirán la construcción de clases derivadas. Java, soporta herencia simple y, por razones de simplicidad, no soporta herencia múltiple, como C++, al objeto de no añadir problemas al diseño. Los conceptos de interfaces también se describen en el capítulo.

Capítulo 5. Algoritmos recursivos. La recursividad es una de las características más sobresalientes en cualquier tipo de programación, Los algoritmos recursivos abundan en la vida ordinaria y el proceso de abstracción que identifica estos algoritmos debe conducir a un buen diseño de algoritmos recursivos. Los algoritmos más sobresalientes y de mayor difusión en el mundo de la programación se explican con detalle en el capítulo. Así, se describen algoritmos como *mergesort* (ordenación por mezclas), *backtracking* (vuelta atrás) y otros.

Capítulo 6. Algoritmos de ordenación y búsqueda. Las operaciones más frecuentes en el proceso de estructura de datos, son: ordenación y búsqueda de datos específicos. Los algoritmos más populares y eficientes de proceso de estructuras de datos internas se describen en el capítulo junto con un análisis de su complejidad. Así, se analizan algoritmos de ordenación básicos y algoritmos avanzados como *Shell, QuickSort, BinSort* o *RadixSort;* en lo relativo a métodos de búsqueda se describen los dos métodos más utilizados: *secuencial* y *binaria*.

Capítulo 7. Algoritmos de ordenación de archivos. Los archivos *(ficheros)* de datos son, posiblemente, las estructuras de datos más diseñadas y utilizadas por los programadores de aplicaciones y programadores de sistemas. Los conceptos de flujos y archivos de Java junto con los métodos clásicos y eficientes de ordenación de archivos se describen en profundidad en este capítulo.

Capítulo 8. Listas enlazadas. Una lista enlazada es una estructura de datos lineal de gran uso en la vida diaria de las personas y de las organizaciones. Su implementación mediante listas enlazadas es el objetivo central de este capítulo. Variantes de las *listas enlazadas simples* como *doblemente enlazadas* y *circulares,* son también, motivo de estudio en el capítulo.

Capítulo 9. Pilas. La pila es una estructura de datos simple cuyo concepto forma también parte, en un elevado porcentaje de la vida diaria de las personas y organizaciones, como las listas. El tipo de dato `Pila` se puede implementar con arrays o con listas enlazadas y describe ambos algoritmos y sus correspondientes implementaciones en Java.

Capítulo 10. Colas. Al igual que las pilas, las colas conforman otra estructura que abunda en la vida ordinaria. La implementación del TAD `Cola` se puede hacer con *arrays* (arreglos), listas enlazadas e incluso listas circulares. Asimismo, se analiza también en el capítulo el concepto de la *bicola* o cola de doble entrada.

Capítulo 11. Colas de prioridades y montículos. Un tipo especial de cola, la cola de prioridades, utilizada en situaciones especiales para la resolución de problemas, y el concepto de montículo *(heap,* en inglés) se analizan detalladamente, junto con un método de ordenación por montículos muy eficiente, sobre todo en situaciones complejas y difíciles.

Capítulo 12. Tablas de dispersión, funciones *hash*. Las tablas aleatorias *hash* junto con los problemas de resolución de colisiones y los diferentes tipos de direccionamiento conforman este capítulo.

Capítulo 13. Árboles. Árboles binarios y árboles ordenados. Los árboles son estructuras de datos no lineales y jerárquicas muy importantes. Estas estructuras son notables en programación avanzada. Los árboles binarios y los árboles binarios de búsqueda se describen con rigor

y profundidad por su importancia en el mundo actual de la programación tanto tradicional (fuera de línea) como en la Web (en línea).

Capítulo 14. Árboles de búsqueda equilibrados. Este capítulo se dedica a la programación avanzada de árboles de búsqueda equilibrada. Estas estructuras de datos son complejas y su diseño y construcción requiere de estrategias y métodos eficientes para su implementación; sin embargo, su uso puede producir grandes mejoras al diseño y construcción de programas que sería muy difícil y por otros métodos.

Capítulo 15. Grafos, representación y operaciones. Los grafos son una de las herramientas más empleadas en matemáticas, estadística, investigación operativa y numerosos campos científicos. El estudio de la teoría de grafos se realiza fundamentalmente como elemento importante de matemática discreta o matemática aplicada. El conocimiento profundo de la teoría de grafos junto con los algoritmos de implementación es fundamental para conseguir el mayor rendimiento de las operaciones con datos, sobre todo si éstos son complejos en su organización.

Capítulo 16. Grafos, algoritmos fundamentales. Un programador de alto nivel no puede dejar de conocer en toda su profundidad la teoría de grafos y sus aplicaciones en el diseño de redes; por estas razones se analizan los algoritmos de *Warshall, Dijkstra* y *Floid* que estudian los caminos mínimos y más cortos. Se termina el capítulo con la descripción del árbol de expansión de coste mínimo y los algoritmos de *Prin* y *Kruskal* que resuelven su diseño e implementación.

Capítulo 17. Colecciones. El importante concepto de colección se estudia en este capítulo. En particular, los *contenedores* e *iteradores* son dos términos imprescindibles para la programación genérica; su conocimiento y diseño son muy importantes en la formación del programador.

El lector puede encontrar en la página web oficial del libro el Anexo con el estudio de *árboles B* y otros *temas avanzados.*

Código Java disponible

Los códigos en Java de todos los programas de este libro están disponibles en la web (Internet) `http://www.mhe.es/joyanes` —en formato Word para que puedan ser utilizados directamente y evitar su "tecleado" en el caso de los programas largos, o bien simplemente, para seleccionar, recortar, modificar… por el lector a su conveniencia, a medida que avanza en su formación.

AGRADECIMIENTOS

Muchos profesores y colegas españoles y latinoamericanos nos han alentado a escribir esta obra, continuación/complemento de nuestra antigua y todavía disponible en librerías, *Estructura de Datos* cuyo enfoque era en el clásico lenguaje Pascal. A todos ellos queremos mostrarles nuestro agradecimiento y, como siempre, brindarles nuestra colaboración si así lo desean.

A los muchos instructores, maestros y profesores, tanto amigos como anónimos, de universidades e institutos tecnológicos y politécnicos de España y Latinoamérica que siempre apoyan nuestras obras y a los que desgraciadamente nunca podremos agradecer individualmente ese apoyo; al menos que conste en este humilde homenaje, nuestro eterno agradecimiento y reconocimiento por ese cariño que siempre prestan a nuestras obras. Como saben aquellos que nos conocen siempre estamos a su disposición en la medida que, físicamente, nos es posible. Gracias a todos, esta obra es posible, en un porcentaje muy alto, por vuestra ayuda y colaboración.

Y como no, a los estudiantes, a los lectores autodidactas y no autodidactas, que siguen nuestras obras. Su apoyo es un gran acicate para seguir nuestra tarea. También, gracias queridos lectores.

Pero si importantes son en esta obra nuestros colegas y lectores españoles y latinoamericanos, no podemos dejar de citar al equipo humano que desde la editorial siempre cuida nuestras obras y sobre todo nos dan consejos, sugerencias, propuestas, nos "soportan" nuestros retrasos, nuestros "cambios" en la redacción, etc. A Carmelo Sánchez, nuestro editor —y, sin embargo, amigo— de McGraw-Hill, que en esta ocasión, para no ser menos, nos ha vuelto a asesorar tanto en la primera fase de realización como en todo el proceso editorial y a nuestro nuevo editor José Luis García que nos ha seguido apoyando y alentando en nuestro trabajo.

Madrid, Mayo de 2007

Contenido

CAPITULO 1

Algoritmos y estructuras de datos

Objetivos

Con el estudio de este capítulo, usted podrá:

- Revisar los conceptos básicos de tipos de datos.
- Introducirse en las ideas fundamentales de estructuras de datos.
- Revisar el concepto de algoritmo y programa.
- Conocer y entender la utilización de la herramienta de programación conocida por “pseudocódigo”.
- Entender los conceptos de análisis, verificación y eficiencia de un algoritmo.
- Conocer las propiedades matemáticas de la notación O.
- Conocer la complejidad de las sentencias básicas de todo programa Java.

Contenido

Conceptos clave

- Algoritmo.
- Complejidad.
- Eficiencia.
- Estructura de datos.
- Notación asintótica.
- Pseudocódigo.
- Rendimiento de un programa.
- Tipo de dato.

INTRODUCCIÓN

La representación de la información es fundamental en ciencias de la computación y en informática. El propósito principal de la mayoría de los programas de computadoras es almacenar y recuperar información, además de realizar cálculos. De modo práctico, los requisitos de almacenamiento y tiempo de ejecución exigen que tales programas deban organizar su información de un modo que soporte procesamiento eficiente. Por estas razones, el estudio de estructuras de datos y de los algoritmos que las manipulan constituye el núcleo central de la *informática* y de la *computación*.

Se revisan en este capítulo los conceptos básicos de *dato, abstracción, algoritmos* y *programas*, así como los criterios relativos a *análisis y eficiencia de algoritmos.*

1.1. TIPOS DE DATOS

Los lenguajes de programación tradicionales, como Pascal y C, proporcionan *tipos de datos* para clasificar diversas clases de datos. La ventajas de utilizar tipos en el desarrollo de software [TREMBLAY 2003] son:

- Apoyo y ayuda en la prevención y en la detección de errores.
- Apoyo y ayuda a los desarrolladores de software, y a la comprensión y organización de ideas acerca de sus objetos.
- Ayuda en la identificación y descripción de propiedades únicas de ciertos tipos.

Ejemplo 1.1

Detección y prevención de errores mediante lenguajes tipificados (tipeados), como Pascal y ALGOL, en determinadas situaciones.

La expresión aritmética

```
6 + 5 + "Ana la niña limeña"
```

contiene un uso inapropiado del literal cadena. Es decir, se ha utilizado un objeto de un cierto tipo (datos enteros) en un contexto inapropiado (datos tipo cadena de caracteres) y se producirá un error con toda seguridad en el momento de la compilación, en consecuencia, en la ejecución del programa del cual forme parte.

Los tipos son un enlace importante entre el mundo exterior y los elementos datos que manipulan los programas. Su uso permite a los desarrolladores limitar su atención a tipos específicos de datos, que tienen propiedades únicas. Por ejemplo, el tamaño es una propiedad determinante en los arrays y en las cadenas; sin embargo, no es una propiedad esencial en los valores lógicos, verdadero `(true)` y falso `(false)`.

Definición 1: Un *tipo de dato* es un conjunto de valores y operaciones asociadas a esos valores.

Definición 2: Un *tipo de dato* consta de dos partes: un conjunto de datos y las operaciones que se pueden realizar sobre esos datos.

En los lenguajes de programación hay disponible un gran número de tipos de datos. Entre ellos se pueden destacar los tipos primitivos de datos, los tipos compuestos y los tipos agregados.

1.1.1. Tipos primitivos de datos

Los tipos de datos más simples son los *tipos de datos primitivos,* también denominados *datos atómicos* porque no se construyen a partir de otros tipos y son entidades únicas no descomponibles en otros.

Un **tipo de dato atómico** es un conjunto de datos atómicos con propiedades idénticas. Estas propiedades diferencian un tipo de dato atómico de otro. Los tipos de datos atómicos se definen por un conjunto de valores y un conjunto de operaciones que actúan sobre esos valores.

Tipo de dato atómico

1. Un conjunto de valores.
2. Un conjunto de operaciones sobre esos valores.

Ejemplo 1.2

Diferentes tipos de datos atómicos

Enteros		
	valores	`-∞ , …, -3, -2, -1, 0, 1, 2, 3, …, +∞`
	operaciones	`*, +, -, /, %, ++, --, …`
Coma flotante		
	valores	`-  , …, 0.0, …,`
	operaciones	`*, +, -, %, /,  …`
Carácter		
	valores	`\0, …, 'A', 'B', …, 'a', 'b', …`
	operaciones	`<, >, …`
Lógico		
	valores	`verdadero, falso`
	operaciones	`and, or, not, …`

Los **tipos numéricos** son, probablemente, los tipos primitivos más fáciles de entender, debido a que las personas están familiarizadas con los números. Sin embargo, los números pueden también ser difíciles de comprender por los diferentes métodos en que son representados en las computadoras. Por ejemplo, los números decimales y binarios tienen representaciones diferentes en las máquinas. Debido a que el número de bits que representa los números es finito, sólo los subconjuntos de enteros y reales se pueden representar. A consecuencia de ello, se pueden presentar situaciones de desbordamiento (*underflow* y *overflow*) positivo y negativo al realizar operaciones aritméticas. Normalmente, los rangos numéricos y la precisión varía de una máquina a otra. Para eliminar estas inconsistencias, algunos lenguajes modernos, como Java y C#, tienen especificaciones precisas para el número de bits, el rango y la precisión numérica de cada operación para cada tipo numérico.

```
0        23       786      456         999
7.56     4.34     0.897    1.23456     99.999
```

El tipo de dato **boolean (lógico)** suele considerarse como el más simple debido a que sólo tiene dos valores posibles: `verdadero (true)` y `falso (false)`. El formato sintáctico de estas constantes lógicas puede variar de un lenguaje de programación a otro. Algunos lenguajes ofrecen un conjunto rico de operaciones lógicas. Por ejemplo, algunos lenguajes tienen construcciones que permiten a los programadores especificar operaciones condicionales o en cortocircuito. Por ejemplo, en Java, en los operadores lógicos `&&` y `||` sólo se evalúa el segundo operando si su valor se necesita para determinar el valor de la expresión global.

El **tipo carácter** consta del conjunto de caracteres disponible para un lenguaje específico en una computadora específica. En algunos lenguajes de programación, un carácter es un símbolo indivisible, mientras que una cadena es una secuencia de cero o más caracteres. Las cadenas se pueden manipular de muchas formas, pero los caracteres implican pocas manipulaciones. Los tipos carácter, normalmente, son dependientes de la máquina.

```
'Q'     'a'     '8'     '9'     'k'
```

Los códigos de carácter más utilizados son EBCDIC (utilizado por las primeras máquinas de IBM) y ASCII (el código universal más extendido). La aparición del lenguaje Java trajo consigo la representación de caracteres en Unicode, un conjunto internacional de caracteres que unifica a muchos conjuntos de caracteres, incluyendo inglés, español, italiano, griego, alemán, latín, hebreo o indio. En Java, el tipo `char` se representa como un valor de 16 bits. Por consiguiente, el rango de `char` está entre `0` y `65.536`, lo que permite representar prácticamente la totalidad de los alfabetos y formatos numéricos y de puntuación universales.

1.1.2. Tipos de datos compuestos y agregados

Los **datos compuestos** son el tipo opuesto a los tipos de datos atómicos. Los datos compuestos se pueden romper en subcampos que tengan significado. Un ejemplo sencillo es el número de su teléfono celular `51199110101`. Realmente, este número consta de varios campos, el código del país (`51`, Perú), el código del área (`1`, Lima) y el número propiamente dicho, que corresponde a un celular porque empieza con `9`.

En algunas ocasiones los datos compuestos se conocen también como datos o tipos agregados. Los **tipos agregados** son tipos de datos cuyos valores constan de colecciones de elementos de datos. Un tipo agregado se compone de tipos de datos previamente definitivos. Existen tres tipos agregados básicos: *arrays* (arreglos), *secuencias* y *registros*.

Un ***array*** o **arreglo**[1] es, normalmente, una colección de datos de tamaño o longitud fija, cada uno de cuyos datos es accesible en tiempo de ejecución mediante la evaluación de las expresiones que representan a los subíndices o índices correspondientes. Todos los elementos de un array deben ser del mismo tipo.

```
array de enteros:   [4, 6, 8, 35, 46, 0810]
```

Una **secuencia** o **cadena** es, en esencia, un array cuyo tamaño puede variar en tiempo de ejecución. Por consiguiente, las secuencias son similares a arrays dinámicos o flexibles.

```
Cadena  = "Aceite picual de Carchelejo"
```

[1] El término inglés *array* se traduce en casi toda Latinoamérica por arreglo, mientras que en España se ha optado por utilizar el término en inglés o bien su traducción por “lista” , “tabla” o “matriz”.

Un **registro** puede contener elementos datos agregados y primitivos. Cada elemento agregado, eventualmente, se descompone en *campos* formados por elementos primitivos. Un registro se puede considerar como un tipo o colección de datos de tamaño fijo. Al contrario que en los arrays, en los que todos sus elementos deben ser del mismo tipo de datos, los campos de los registros pueden ser de diferentes tipos de datos. A los campos de los registros se accede mediante identificadores.

El registro es el tipo de dato más próximo a la idea de objeto. En realidad, el concepto de objeto en un desarrollo orientado a objetos es una generalización del tipo registro.

```
Registro   {
              Dato1
              Dato2
              Dato3
               ...
           }
```

1.2. LA NECESIDAD DE LAS ESTRUCTURAS DE DATOS

A pesar de la gran potencia de las computadoras actuales, la eficiencia de los programas sigue siendo una de las características más importantes a considerar. Los problemas complejos que procesan las computadoras cada vez más obligan, sobre todo, a pensar en su eficiencia dado el elevado tamaño que suelen alcanzar. Hoy, más que nunca, los profesionales deben formarse en técnicas de construcción de programas eficientes.

En sentido general, una estructura de datos es cualquier representación de datos y sus operaciones asociadas. Bajo esta óptica, cualquier representación de datos, incluso un número entero o un número de coma flotante almacenado en la computadora, es una sencilla estructura de datos. En un sentido más específico, una estructura de datos es una organización o estructuración de una colección de elementos dato. Así, una lista ordenada de enteros almacenados en un array es un ejemplo de dicha estructuración.

Una **estructura de datos** es una agregación de tipos de datos compuestos y atómicos en un conjunto con relaciones bien definidas. Una estructura significa un conjunto de reglas que contienen los datos juntos.

Las estructuras de datos pueden estar *anidadas:* se puede tener una estructura de datos que conste de otras.

Estructura de datos

1. Una combinación de elementos en la que cada uno es o bien un tipo de dato u otra estructura de datos.
2. Un conjuntos de asociaciones o relaciones (estructura) que implica a los elementos combinados.

La Tabla 1.1 recoge las definiciones de dos estructuras de datos clásicas: *arrays* y registros.

Tabla 1.1 Ejemplos de estructura de datos

Array	Registro
Secuencias homogéneas de datos o tipos de datos conocidos como elementos.	Combinación de datos heterogéneos en una estructura única con una clave identificada.
Asociación de posición entre los elementos.	Ninguna asociación entre los elementos.

La mayoría de los lenguajes de programación soporta diferentes estructuras de datos. Además, esos mismos lenguajes suelen permitir a los programadores crear sus propias nuevas estructuras de datos con el objetivo fundamental de resolver del modo más eficiente posible una aplicación.

Será necesario que la elección de una estructura de datos adecuada requiera también la posibilidad de poder realizar operaciones sobre dichas estructuras. La elección de la estructura de datos adecuada redundará en una mayor eficiencia del programa y, sobre todo, en una mejor resolución del problema en cuestión. Una elección inadecuada de la estructura de datos puede conducir a programas lentos, largos y poco eficientes.

Una solución se denomina **eficiente** si resuelve el problema dentro de las *restricciones de recursos requeridas.* Restricciones de recursos pueden ser el espacio total disponible para almacenar los datos (considerando la memoria principal independiente de las restricciones de espacio de discos, fijos, CD, DVD, *flash*…) o el tiempo permitido para ejecutar cada subtarea. Se suele decir que una solución es eficiente cuando requiere menos recursos que las alternativas conocidas.

El **costo** de una solución es la cantidad de recursos que la solución consume. Normalmente, el coste se mide en término de recursos clave, especialmente el tiempo.

1.2.1. Etapas en la selección de una estructura de datos

Los pasos a seguir para seleccionar una estructura de datos que resuelva un problema son [SHAFFER 97]:

1. Analizar el problema para determinar las restricciones de recursos que debe cumplir cada posible solución.
2. Determinar las operaciones básicas que se deben soportar y cuantificar las restricciones de recursos para cada una. Ejemplos de operaciones básicas son la inserción de un dato en la estructura de datos, suprimir un dato de la estructura o encontrar un dato determinado en dicha estructura.
3. Seleccionar la estructura de datos que cumple mejor los requisitos o requerimientos.

Este método de tres etapas para la selección de una estructura de datos es una aproximación centrada en los datos. Primero se diseñan los datos y las operaciones que se realizan sobre ellos, a continuación viene la representación de esos datos y, por último, viene la implementación de esa representación.

Las restricciones de recursos sobre ciertas operaciones clave, como búsqueda, inserción de registros de datos y eliminación de registros de datos, normalmente conducen el proceso de selección de las estructuras de datos.

Algunas consideraciones importantes para la elección de la estructura de datos adecuada son:

- ¿Todos los datos se insertan en la estructura de datos al principio o se entremezclan con otras operaciones?
- ¿Se pueden eliminar los datos?
- ¿Los datos se procesan en un orden bien definido o se permite el acceso aleatorio?

RESUMEN

Un tipo es una colección de valores. Por ejemplo, el tipo boolean consta de los valores *true* y *false*. Los enteros también forman un tipo.

Un tipo de dato es un tipo con una colección de operaciones que manipulan el tipo. Por ejemplo, una variable entera es un miembro del tipo de dato entero. La suma es un ejemplo de una operación sobre tipos de datos enteros.

Un elemento dato es una pieza de información o un registro cuyos valores se especifican a partir de un tipo. Un elemento dato se considera que es un miembro de un tipo de dato. El entero es un elemento de datos simple ya que no contiene subpartes.

Un registro de una cuenta corriente de un banco puede contener varios campos o piezas de información como nombre, número de la cuenta, saldo y dirección. Dicho registro es un dato agregado.

1.3. ALGORITMOS Y PROGRAMAS

Un **algoritmo** es un método, un proceso, un conjunto de instrucciones utilizadas para resolver un problema específico. Un problema puede ser resuelto mediante muchos algoritmos. Un algoritmo dado correcto, resuelve un problema definido y determinado (por ejemplo, calcula una función determinada). En este libro se explican muchos algoritmos y, para algunos problemas, se proponen diferentes algoritmos, como es el caso del problema típico de ordenación de listas.

La ventaja de conocer varias soluciones a un problema es que las diferentes soluciones pueden ser más eficientes para variaciones específicas del problema o para diferentes entradas del mismo problema. Por ejemplo, un algoritmo de ordenación puede ser el mejor para ordenar conjuntos pequeños de números, otro puede ser el mejor para ordenar conjuntos grandes de números y un tercero puede ser el mejor para ordenar cadenas de caracteres de longitud variable.

Desde un punto de vista más formal y riguroso, un algoritmo es "un conjunto ordenado de pasos o instrucciones ejecutables y no ambiguas". Obsérvese en la definición que las etapas o pasos que sigue el algoritmo deben tener una estructura bien establecida en términos del orden en que se ejecutan las etapas. Esto no significa que las etapas se deban ejecutar en secuencia: una primera etapa, después una segunda, etc. Algunos algoritmos, conocidos como *algoritmos paralelos,* por ejemplo, contienen más de una secuencia de etapas, cada una diseñada para ser ejecutada por procesadores diferentes en una máquina multiprocesador. En tales casos, los algoritmos globales no poseen un único hilo conductor de etapas que conforman el escenario de primera etapa, segunda etapa, etc. En su lugar, la estructura del algoritmo es el de múltiples hilos conductores que se bifurcan y se reconectan a medida que los diferentes procesadores ejecutan las diferentes partes de la tarea global.

Durante el diseño de un algoritmo, los detalles de un lenguaje de programación específico se pueden obviar frente a la simplicidad de una solución. Generalmente, el diseño se escribe en español (o en inglés, o en otro idioma hablado). También se utiliza un tipo de lenguaje mixto entre el español y un lenguaje de programación universal que se conoce como ***pseudocódigo*** (o *seudocódigo*).

1.3.1. Propiedades de los algoritmos

Un algoritmo debe cumplir diferentes propiedades[2]:

1. *Especificación precisa de la entrada.* La forma más común del algoritmo es una transformación que toma un conjunto de valores de entrada y ejecuta algunas manipulaciones para producir un conjunto de valores de salida. Un algoritmo debe dejar claros el número y tipo de valores de entrada y las condiciones iniciales que deben cumplir esos valores de entrada para conseguir que las operaciones tengan éxito.
2. *Especificación precisa de cada instrucción.* Cada etapa de un algoritmo debe ser definida con precisión. Esto significa que no puede haber *ambigüedad* sobre las acciones que se deban ejecutar en cada momento.
3. *Exactitud, corrección.* Un algoritmo debe ser *exacto, correcto.* Se debe poder demostrar que el algoritmo resuelve el problema. Con frecuencia, esto se plasma en el formato de un argumento, lógico o matemático, al efecto de que ***si*** las condiciones de entrada se cumplen y se ejecutan los pasos del algoritmo, ***entonces*** se producirá la salida deseada. En otras palabras, se debe calcular la función deseada y convertir cada entrada a la salida correcta. Un algoritmo se espera que resuelva un problema.
4. *Etapas bien definidas y concretas.* Un algoritmo se compone de una serie de *etapas concretas,* lo que significa que la acción descrita por esa etapa está totalmente comprendida por la persona o máquina que debe ejecutar el algoritmo. Cada etapa debe ser ejecutable en una cantidad finita de tiempo. Por consiguiente, el algoritmo nos proporciona una "receta" para resolver el problema en etapas y tiempos concretos.
5. *Número finito de pasos.* Un algoritmo se debe componer de un número *finito* de pasos. Si la descripción del algoritmo consta de un número infinito de etapas, nunca se podrá implementar como un programa de computador. La mayoría de los lenguajes que describen algoritmos (español, inglés o pseudocógio) proporciona un método para ejecutar acciones repetidas, conocidas como iteraciones, que controlan las salidas de bucles o secuencias repetitivas.
6. *Un algoritmo debe terminar.* En otras palabras, no puede entrar en un bucle infinito.
7. *Descripción del resultado o efecto.* Por último, debe estar claro cuál es la tarea que el algoritmo debe ejecutar. La mayoría de las veces, esta condición se expresa con la producción de un valor como resultado que tenga ciertas propiedades. Con menor frecuencia, los algoritmos se ejecutan para un *efecto lateral,* como imprimir un valor en un dispositivo de salida. En cualquier caso, la salida esperada debe estar especificada completamente.

Ejemplo 1.3

¿Es un algoritmo la instrucción siguiente?

```
Escribir una lista de todos los enteros positivos
```

Es imposible ejecutar la instrucción anterior dado que hay infinitos enteros positivos. Por consiguiente, cualquier conjunto de instrucciones que implique esta instrucción no es un algoritmo.

[2] En [BUDD 1998], [JOYANES 2003], [BROOKSHEAR 2003] y [TREMBLAY 2003], además de en las referencias incluidas al final del libro en la bibliografía, puede encontrar información ampliada sobre teoría y práctica de algoritmos.

1.3.2. Programas

Normalmente, se considera que un programa de computadora es una representación concreta de un algoritmo en un lenguaje de programación. Naturalmente, hay muchos programas que son ejemplos del mismo algoritmo, dado que cualquier lenguaje de programación moderno se puede utilizar para implementar cualquier algoritmo (aunque algunos lenguajes de programación facilitarán su tarea al programador más que otros). Por definición un algoritmo debe proporcionar suficiente detalle para que se pueda convertir en un programa cuando se necesite.

El requisito de que un algoritmo "debe terminar" significa que no todos los programas de computadora son algoritmos. Su sistema operativo es un programa en tal sentido; sin embargo, si piensa en las diferentes tareas de un sistema operativo (cada una con sus entradas y salidas asociadas) como problemas individuales, cada una es resuelta por un algoritmo específico implementado por una parte del programa sistema operativo, cada una de las cuales termina cuando se produce su correspondiente salida.

Para recordar

1. Un problema es una función o asociación de entradas con salidas.
2. Un algoritmo es una receta para resolver un problema cuyas etapas son concretas y no ambiguas.
3. El algoritmo debe ser correcto y finito, y debe terminar para todas las entradas.
4. Un programa es una "ejecución" *(instanciación)* de un algoritmo en un lenguaje de programación de computadora.

El diseño de un algoritmo para ser implementado por un programa de computadora debe tener dos características principales:

1. Que sea fácil de entender, codificar y depurar.
2. Que consiga la mayor eficiencia para los recursos de la computadora.

Idealmente, el programa resultante debería ser el más eficiente. ¿Cómo medir la eficiencia de un algoritmo o programa? El método correspondiente se denomina *análisis de algoritmos* y permite medir la dificultad inherente a un problema. En este capítulo se desarrollará este concepto y la forma de medir la medida de la eficiencia.

1.4. EFICIENCIA Y EXACTITUD

De las características que antes se han analizado y deben cumplir los algoritmos, destacan dos por su importancia en el desarrollo de algoritmos y en la construcción de programas: eficiencia y exactitud, que se examinarán y utilizarán amplia y profusamente en los siguientes capítulos.

Existen numerosos enfoques a la hora de resolver un problema. ¿Cómo elegir el más adecuado entre ellos? Entre las líneas de acción fundamentales en el diseño de computadoras se suelen plantear dos objetivos (a veces conflictivos y contradictorios entre sí) [SHAFFER 1997]:

1. Diseñar un algoritmo que sea fácil de entender, codificar y depurar.
2. Diseñar un algoritmo que haga un uso eficiente de los recursos de la computadora.

Idealmente, el programa resultante debe cumplir ambos objetivos. En estos casos, se suele decir que dicho programa es "elegante". Entre los objetivos centrales de este libro está la medida de la eficiencia de algoritmo, así como su diseño correcto o exacto.

1.4.1. Eficiencia de un algoritmo

Raramente existe un único algoritmo para resolver un problema determinado. Cuando se comparan dos algoritmos diferentes que resuelven el mismo problema, normalmente se encontrará que un algoritmo es un orden de magnitud más eficiente que el otro. En este sentido, lo importante es que el programador sea capaz de reconocer y elegir el algoritmo más eficiente.

Entonces, ¿qué es eficiencia? La ***eficiencia*** de un algoritmo es la propiedad mediante la cual un algoritmo debe alcanzar la solución al problema en el tiempo más corto posible o utilizando la cantidad más pequeña posible de recursos físicos y que sea compatible con su exactitud o corrección. Un buen programador buscará el algoritmo más eficiente dentro del conjunto de aquellos que resuelven con exactitud un problema dado.

¿Cómo medir la eficiencia de un algoritmo o programa informático? Uno de los métodos más sobresalientes es el ***análisis de algoritmos,*** que permite medir la dificultad inherente de un problema. Los restantes capítulos utilizan con frecuencia las técnica de análisis de algoritmos siempre que estos se diseñan. Esta característica le permitirá comparar algoritmos para la resolución de problemas en términos de eficiencia.

Aunque las máquinas actuales son capaces de ejecutar millones de instrucciones por segundo, la eficiencia permanece como un reto o preocupación a resolver. Con frecuencia, la elección entre algoritmos eficientes e ineficientes pueden mostrar la diferencia entre una solución práctica a un problema y una no práctica. En los primeros tiempos de la informática moderna (décadas de los 60 a los 80), las computadoras eran muy lentas y tenían una capacidad de memoria pequeña. Los programas tenían que ser diseñados cuidadosamente para hacer uso de los recursos escasos, como almacenamiento y tiempo de ejecución. Los programadores gastaban horas intentando recortar radicalmente segundos a los tiempos de ejecución de sus programas o intentando comprimir los programas en un pequeño espacio en memoria utilizando todo tipo de tecnologías de comprensión y reducción de tamaño. La eficiencia de un programa se medía en aquella época como un factor dependiente del binomio *espacio-tiempo.*

Hoy, la situación ha cambiado radicalmente. Los costes del hardware han caído drásticamente, mientras que los costes humanos han aumentado considerablemente. El tiempo de ejecución y el espacio de memoria ya no son factores críticos como lo fueron anteriormente. Hoy día, el esfuerzo considerable que se requería para conseguir la eficiencia máxima no es tan acusado, excepto en algunas aplicaciones como, por ejemplo, sistemas en tiempo real con factores críticos de ejecución. Pese a todo, la eficiencia sigue siendo un factor decisivo en el diseño de algoritmos y construcción posterior de programas.

Existen diferentes métodos con los que se trata de medir la eficiencia de los algoritmos; entre ellos, los que se basan en el número de operaciones que debe efectuar un algoritmo para realizar una tarea; otros métodos se centran en tratar de medir el tiempo que se emplea en llevar a cabo una determinada tarea, ya que lo importante para el usuario final es que ésta se efectúe de forma correcta y en el menor tiempo posible. Sin embargo, estos métodos presentan varias dificultades, ya que cuando se trata de generalizar la medida hecha, ésta depende de factores como la máquina en la que se efectuó, el ambiente del procesamiento y el tamaño de la muestra, entre otros factores.

Brassard y *Bratley* acuñaron, en 1988, el término **algoritmia** *(algorithmics)*[3], que definía como “el estudio sistemático de las técnicas fundamentales utilizadas para diseñar y analizar algoritmos eficientes”. Este estudio fue ampliado en 1997[4] con la consideración de que la determinación de la eficiencia de un algoritmo se podía expresar en el tiempo requerido para realizar la tarea en función del tamaño de la muestra e independiente del ambiente en que se efectúe.

El estudio de la eficiencia de los algoritmos se centra, fundamentalmente, en el análisis de la ejecución de bucles, ya que en el caso de funciones lineales —no contienen bucles—, la eficiencia es función del número de instrucciones que contiene. En este caso, su eficiencia depende de la velocidad de las computadoras y, generalmente, no es un factor decisivo en la eficiencia global de un programa.

Al crear programas que se ejecutan muchas veces a lo largo de su vida y/o tienen grandes cantidades de datos de entrada, las consideraciones de eficiencia, no se pueden descartar. Además, en la actualidad existen un gran número de aplicaciones informáticas que requieren características especiales de *hardware* y *software* en las cuales los criterios de eficiencia deben ser siempre tenidas en cuenta.

Por otra parte, las consideraciones espacio-tiempo se han ampliado con los nuevos avances en tecnologías de hardware de computadoras. Las consideraciones de espacio implican hoy diversos tipos de memoria: principal, *caché, flash,* archivos, discos duros USB y otros formatos especializados. Asimismo, con el uso creciente de redes de computadoras de alta velocidad, existen muchas consideraciones de eficiencia a tener en cuenta en entornos de informática o computación distribuida.

Nota

La eficiencia como factor espacio-tiempo debe estar estrechamente relacionada con la buena calidad, el funcionamiento y la facilidad de mantenimiento del programa.

1.4.2. Formato general de la eficiencia

En general, el formato se puede expresar mediante una función:

$f(n) = eficiencia$

Es decir, la eficiencia del algoritmo se examina como una función del número de elementos que tienen que ser procesados.

Bucles lineales

En los bucles se repiten las sentencias del *cuerpo del bucle* un número determinado de veces, que determina la eficiencia del mismo. Normalmente, en los algoritmos los bucles son el término dominante en cuanto a la eficiencia del mismo.

---

[3] Giles Brassard y Paul Bratley. *Algorithmics. Theory and Practice*. Englewood Cliffs, N. N.: Prentice-Hall, 1988.

[4] *Fundamental of algorithmics* (Prentice-Hall, 1997). Este libro fue traducido al español, publicado también en Prentice-Hall (España) por un equipo de profesores de la Universidad Pontificia de Salamanca, dirigidos por el co-autor de este libro, el profesor Luis Joyanes.

Ejemplo 1.4

¿Cuántas veces se repite el cuerpo del bucle en el siguiente código?

```
1  i = 1
2  iterar (i <= n)
      1 código de la aplicación
      2 i = i + 1
3  fin _ iterar
```

Si `n` es un entero, por ejemplo de valor `100`, la respuesta es `100` veces. El número de iteraciones es directamente proporcional al factor del bucle, `n`. Como la eficiencia es directamente proporcional al número de iteraciones, la función que expresa la eficiencia es:

```
f(n) =  n
```

Ejemplo 1.5

¿Cuántas veces se repite el cuerpo del bucle en el siguiente código?

```
1  i = 1
2  iterar (i <= n)
      1 código de la aplicación
      2 i = i + 2
3  fin _ iterar
```

La respuesta no siempre es tan evidente como en el ejercicio anterior. Ahora el contador `i` avanza de `2` en `2`, por lo que la respuesta es `n/2`. En este caso, el factor de eficiencia es:

```
f(n) =  n / 2
```

Bucles algorítmicos

Consideremos un bucle en el que su variable de control se multiplique o divida dentro de dicho bucle. ¿Cuántas veces se repetirá el cuerpo del bucle en los siguientes segmentos de programa?

```
1  i = 1
2  mientras (i < 1000)
      { código de la aplicación }
      i = i * 2
3  fin _ mientras
```

```
1  i = 1000
2  mientras (i >= 1)
     { código aplicación }
     i = i/2
3  fin _ mientras
```

La Tabla 1.2 contiene las diferentes iteraciones y los valores de la variable `i`.

Tabla 1.2 Análisis de los bucles de multiplicación y división

Bucle de multiplicar		Bucle de dividir	
Iteración	Valor de i	Iteración	Valor de i
1	1	1	1000
2	2	2	500

(Continúa)

Bucle de multiplicar		Bucle de dividir	
Iteración	Valor de i	Iteración	Valor de i
3	4	3	250
4	8	4	125
5	16	5	62
6	32	6	31
7	64	7	15
8	128	8	7
9	256	9	3
10	512	10	1
salida	1024	*salida*	0

En ambos bucles se ejecutan 10 iteraciones. La razón es que, en cada iteración, el valor de `i` se dobla en el bucle de multiplicar y se divide por la mitad en el bucle de división. Por consiguiente, el número de iteraciones es una función del multiplicador o divisor, en este caso 2.

Bucle de multiplicar $2^{iteraciones} < 1000$
Bucle de división $1000 / 2^{iteraciones} >= 1$

Generalizando el análisis, se puede decir que las iteraciones de los bucles especificados se determinan por la siguiente fórmula:

$$f(n) = [\log_2 n]$$

Bucles anidados

En el caso de bucles anidados (bucles que contienen otros bucles), se determinan cuántas iteraciones contiene cada bucle. El total es entonces el producto del número de iteraciones del bucle interno y el número de iteraciones del bucle externo.

```
iteraciones : iteraciones del bucle externo x iteraciones bucle interno
```

Existen tres tipos de bucles anidados: *lineal logarítmico, cuadrático dependiente* y *cuadrático* que con ejemplos y análisis similares a las anteriores nos conducen a ecuaciones de eficiencia contempladas en la Tabla 1.3.

Tabla 1.3 Fórmulas de eficiencia

Lineal logarítmica	Dependiente cuadrática	Cuadrática
$f(n) = [n\log_2 n]$	$f(n) = \frac{n (n + 1)}{2}$	$f(n) = n^2$

1.4.3. Análisis de rendimiento

La medida del rendimiento de un programa se consigue mediante la complejidad del espacio y del tiempo de un programa.

La *complejidad del espacio* de un programa es la cantidad de memoria que se necesita para ejecutarlo hasta la compleción (*terminación).* El avance tecnológico proporciona hoy en día memoria abundante; por esa razón, el análisis de algoritmos se centra fundamentalmente en el tiempo de ejecución.

La *complejidad del tiempo* de un programa es la cantidad de tiempo de computadora que se necesita para ejecutarlo. Se utiliza una función, $T(n)$, para representar el número de unidades de tiempo tomadas por un programa o algoritmo para cualquier entrada de tamaño n. Si la función $T(n)$ de un programa es $T(n) = c * n$, entonces el tiempo de ejecución es linealmente proporcional al tamaño de la entrada sobre la que se ejecuta. Tal programa se dice que es de *tiempo lineal* o, simplemente *lineal*.

Ejemplo 1.6

Tiempo de ejecución lineal de una función que calcula una serie de n términos.

```
double serie(double x, int n)
{
   double s;
   int i;
   s = 0.0;                        // tiempo t1
   for (i = 1; i <= n; i++)        // tiempo t2
   {
      s += i*x;                    // tiempo t3
   }
   return s;                       // tiempo t4
}
```

La función $T(n)$ del método es:

```
T(n) = t1 + n*t2 + n*t3 + t4
```

El tiempo crece a medida que lo hace n, por lo que es preferible expresar el tiempo de ejecución de tal forma que indique el comportamiento que va a tener la función con respecto al valor de n.

Considerando todas las reflexiones anteriores, si $T(n)$ es el tiempo de ejecución de un programa con entrada de tamaño n, será posible valorar $T(n)$ como el número de sentencias, en nuestro caso en Java, ejecutadas por el programa, y la evaluación se podrá efectuar desde diferentes puntos de vista:

Peor caso. Indica el tiempo peor que se puede tener. Este análisis es perfectamente adecuado para algoritmos cuyo tiempo de respuesta es crítico; por ejemplo, para el caso del programa de control de una central nuclear. Es el que se emplea en este libro.

Mejor caso. Indica el tiempo mejor que podemos tener.

Caso medio. Se puede computar $T(n)$ como el tiempo medio de ejecución del programa sobre todas las posibles ejecuciones de entradas de tamaño n. El tiempo de ejecución medio es a veces una medida más realista de lo que el rendimiento será en la práctica, pero es, normalmente, mucho más difícil de calcular que el tiempo de ejecución en el peor caso.

1.5. NOTACIÓN O-GRANDE

La alta velocidad de las computadoras actuales (frecuencias del procesador de 3 GHz ya son usuales en computadoras comerciales) hace que la medida exacta de la eficiencia de un algoritmo no sea una preocupación vital en el diseño, pero sí el orden general de magnitud de la misma. Si el análisis de dos algoritmos muestra que uno ejecuta 25 iteraciones mientras otro ejecuta 40, la práctica muestra que ambos son muy rápidos; entonces, ¿cómo se valoran las diferencias? Por otra parte, si un algoritmo realiza 25 iteraciones y otro 2.500 iteraciones, entonces debemos estar preocupados por la rapidez de uno o la lentitud de otro.

Se ha comentado anteriormente que el número de sentencias ejecutadas en la función para `n` datos es función del número de elementos y se expresa por la fórmula *f(n)*. Aunque la ecuación para obtener una función puede ser compleja, el factor dominante que se debe considerar para determinar el orden de magnitud del resultado es el denominado factor de eficiencia. Por consiguiente, no se necesita determinar la medida completa de la eficiencia, basta con calcular el factor que determina la magnitud. Este factor se define como **"O grande",** que representa *"está en el orden de"* y se expresa como *O(n),* es decir, *"en el orden de n"*.

La notación *O* indica la cota superior del tiempo de ejecución de un algoritmo o programa. Así, en lugar de decir que un algoritmo emplea un tiempo de *4n-1* en procesar un array de longitud *n*, se dirá que emplea un tiempo *O(n)* que se lee *"O grande de n"*, o bien *"O de n"* y que informalmente significa *"algunos tiempos constantes n"*.

> Con la notación O se expresa una aproximación de la relación entre el tamaño de un problema y la cantidad de proceso necesario para hacerlo. Por ejemplo, si
>
> $f(n) = n^2 - 2n + 3$ entonces $f(n)$ es $O(n^2)$.

1.5.1. Descripción de tiempos de ejecución con la notación O

Sea *T(n)* el tiempo de ejecución de un programa, medido como una función de la entrada de tamaño *n*. Se dice que *"T(n) es O(g(n))"* si *g(n)* acota superiormente a *T(n)*. De modo más riguroso, *T(n)* es *O(g(n))* si existe un entero n_0 y una constante $c > 0$ tal que para todos los enteros $n \geq n_o$ se tiene que $T(n) \leq cg(n)$.

Ejemplo 1.7

Dada la función $f(n) = n^3 + 3n + 1$*, encontrar su* "`O grande`" *(complejidad asintótica).*

Para valores de $n \geq 1$ se puede demostrar que:

$$f(n) = n^3 + 3n + 1 \leq n^3 + 3n^3 + 1n^3 = 5n^3$$

Escogiendo la constante $c = 5$ y $n_0 = 1$ se satisface la desigualdad $f(n) \leq 5n^3$. Entonces se puede asegurar que:

$$f(n) = O(n^3)$$

Ahora bien, también se puede asegurar que $f(n) = O(n^4)$ y que $f(n) = O(n^5)$, y así sucesivamente con potencias mayores de `n`. Sin embargo, lo que realmente interesa es la cota superior más ajustada que informa de la tasa de crecimiento de la función con respecto a `n`.

Una función f(x) puede estar acotada superiormente por un número indefinido de funciones a partir de ciertos valores x_0,

$$f(x) \leq g(x) \leq h(x) \leq k(x) \ \ldots$$

La complejidad asintótica de la función f(x) se considera que es la cota superior mas ajustada:

$$f(x) = O(g(x))$$

Para recordar

La expresión la eficiencia de un algoritmo se simplifica con la función *O* grande.

Si un algoritmo es cuadrático, se dice entonces que su eficiencia es $O(n^2)$. Esta función expresa cómo crece el tiempo de proceso del algoritmo, de tal forma que una eficiencia $O(n^2)$ muestra que crece con el cuadrado del número de entradas.

1.5.2. Determinar la notación O

La notación *O* grande se puede obtener de `f(n)` utilizando los siguientes pasos:

1. En cada término, establecer el coeficiente del término en 1.
2. Mantener el término mayor de la función y descartar los restantes. Los términos se ordenan de menor a mayor:

$$\log_2 n \quad n \quad n\log_2 n \quad n^2 \quad n^3 \quad \ldots \quad n^k \quad 2^n \quad n!$$

Ejemplo 1.8

Calcular la función O grande de eficiencia de las siguientes funciones:

a. $F(n) = 1/2n(n+1) = 1/2n^2 + 1/2n$

b. $F(n) = a_j n^k + a_{j-1} n^{k-1} + \ldots + a_2 n^2 + a_1 n + a_0$

Caso a.

1. Se eliminan todos los coeficientes y se obtiene

$$n^2 + n$$

2. Se eliminan los factores más pequeños

$$n^2$$

3. La notación *O* correspondiente es

$$O(f(n)) = O(n^2)$$

Caso b.

1. Se eliminan todos los coeficientes y se obtiene

   $f(n) = n^k + n^{k-1} + ... + n^2 + n + 1$

3. Se eliminan los factores más pequeños y el término de exponente mayor es el primero

   n^k

4. La notación *O* correspondiente es

   $O(f(n)) = O(n^k)$

1.5.3. Propiedades de la notación O-grande

De la definición conceptual de la notación *O* se deducen las siguientes propiedades de la notación *O*.

1. Siendo *c* una constante, $c*O(f(n)) = O(f(n))$.

   Por ejemplo, si $f(n) = 3n^4$, entonces $f(n) = 3*O(n^4) = O(n^4)$

2. $O(f(n)) + O(g(n)) = O(f(n)+g(n))$.

   Por ejemplo, si $f(n) = 2e^n$ y $g(n) = 2n^3$:

   $O(f(n)) + O(g(n)) = O(f(n)+g(n)) = O(2e^n + 2n^3) = O(e^n)$

3. *Maximo(O(f(n)),O(g(n))) = O(Maximo(f(n),g(n))*.

   Por ejemplo,

   Maximo(O(log(n)),O(n)) = O(Maximo(log(n),n)) = O(n)

4. $O(f(n)) * O(g(n)) = O(f(n)*g(n))$.

   Por ejemplo, si $f(n) = 2n^3$ y $g(n) = n$:

   $O(f(n))* O(g(x)) = O(f(x)*g(x)) = O(2n^3 *n) = O(n^4)$

5. $O(log_a(n)) = O(log_b(n))$ para $a, b > 1$.

   Las funciones logarítmicas son de orden logarítmico, independientemente de la base del logaritmo.

6. $O(log(n!)) = O(n*log(n))$.

7. Para $k > 1$ $O(\sum_{i=1}^{n} i^k) = O(n^{k+1})$.

8. $O(\sum_{i=2}^{n} \log(i) = O(n \log(n))$.

1.6. COMPLEJIDAD DE LAS SENTENCIAS BÁSICAS DE JAVA

Al analizar la complejidad de un método no recursivo, se han de aplicar las propiedades de la notación *O* y las siguientes consideraciones relativas al orden que tienen las sentencias, fundamentalmente a las estructuras de control.

- Las sentencias de asignación, son de orden constante *O(1)*.
- La complejidad de una sentencia de selección es el máximo de las complejidades del bloque `then` y del bloque `else`.
- La complejidad de una sentencia de selección múltiple (`switch`) es el máximo de las complejidades de cada uno de los bloques `case`.
- Para calcular la complejidad de un bucle, condicional o automático, se ha de estimar el número máximo de iteraciones para el *peor caso;* entonces, la complejidad del bucle es el producto del número de iteraciones por la complejidad de las sentencias que forman el cuerpo del bucle.
- La complejidad de un bloque se calcula como la suma de las complejidades de cada sentencia del bloque.
- La complejidad de la llamada a un método es de orden 1, complejidad constante. Es necesario considerar la complejidad del método invocado.

Ejemplo 1.9

Determinar la complejidad del método:

```
double mayor(double x, double y)
{
   if (x > y)
      return x;
   else
      return y;
}
```

El método consta de una sentencia de selección, cada alternativa tiene complejidad constante, *O(1);* entonces, la complejidad del método `mayor()` es *O(1).*

Ejemplo 1.10

Determinar la complejidad del siguiente método:

```
void escribeVector(double[] x, int n)
{
   int j;
   for (j = 0; j < n; j++)
   {
      System.out.println(x[j]);
   }
}
```

El método consta de un bucle que se ejecuta *n* veces, *O(n).* El cuerpo del bucle es la llamada a `println()`, complejidad constante *O(1).* Como conclusión, la complejidad del método es `O(n)*O(1) = O(n)`.

Ejemplo 1.11

Determinar la complejidad del método:

```
double suma(double[]d, int n)
{
   int k ; double s;
   k = s = 0;
   while (k < n)
   {
      s += d[k];
      if (k == 0)
         k = 2;
      else
         k = 2 * k;
   }
   return s;
}
```

`suma()` está formado por una sentencia de asignación múltiple, *O(1),* de un bucle condicional y la sentencia que devuelve control de complejidad constante, *O(1).* Por consiguiente, la complejidad del método está determinada por el bucle. Es necesario determinar el número máximo de iteraciones que va a realizar el bucle y la complejidad de su cuerpo. El número de iteraciones es igual al número de veces que el algoritmo multiplica por dos a la variable `k`. Si *t* es el número de veces que `k` se puede multiplicar hasta alcanzar el valor de `n`, que hace que termine el bucle, entonces $k = 2^t$.

$$0,\ 2,\ 2^2,\ 2^3,\ \ldots\ ,\ 2^t \geq n.$$

Tomando logaritmos: $t \geq \log_2 n$; por consiguiente, el máximo de iteraciones es $t = \log_2 n$.

El cuerpo del bucle consta de una sentencia simple y un sentencia de selección de complejidad *O(1)*, por lo que tiene complejidad constante, *O(1).* Con todas estas consideraciones, la complejidad del bucle y del método es:

$O(\log_2 n)*O(1) = O(\log_2 n)$; *complejidad logarítmica* $O(\log n)$

Ejemplo 1.12

Determinar la complejidad del método:

```
void traspuesta(float[][] d, int n)
{
   int i, j;
   for (i = n - 2; i > 0; i--)
   {
      for (j = i + 1; j < n; j++)
      {
         float t;
         t = d[i][j];
         d[i][j] = d[j][i];
         d[j][i] = t;
      }
   }
}
```

El método consta de dos bucles `for` anidados. El bucle interno está formado por tres sentencias de complejidad constante, *O(1).* El bucle externo siempre realiza `n-1` veces el bucle interno. A su vez, el bucle interno hace *k* veces su bloque de sentencias, *k* varía de 1 a *n-1,* de modo que el número total de iteraciones es:

$$C = \sum_{k=1}^{n-1} k$$

El desarrollo del sumatorio produce la expresión:

$$(n-1) + (n-2) + \ldots + 1 = \frac{n*(n-1)}{2}$$

Aplicando las propiedades de la notación *O,* se deduce que la complejidad del método es $O(n^2)$. El término que domina en el tiempo de ejecución es n^2, se dice que la *complejidad es cuadrática*.

RESUMEN

Una de las herramientas típicas más utilizadas para definir algoritmos es el pseudocódigo. El *pseudocódigo* es una representación en español (o en inglés, brasilero, etc.) del código requerido para un algoritmo.

Los datos atómicos son datos simples que no se pueden descomponer. Un tipo de dato atómico es un conjunto de datos atómicos con propiedades idénticas. Este tipo de datos se definen por un conjunto de valores y un conjunto de operaciones que actúa sobre esos valores.

Una estructura de datos es un agregado de datos atómicos y datos compuestos en un conjunto con relaciones bien definidas.

La eficiencia de un algoritmo se define, generalmente, en función del número de elementos a procesar y el tipo de bucle que se va a utilizar. Las eficiencias de los diferentes bucles son:

Bucle lineal:	f(n) = n
Bucle logarítmico:	f(n) = log n
Bucle logarítmico lineal:	f(n) = n * log n
Bucle cuadrático dependiente:	f(n) = n(n+1)/2
Bucle cuadrático dependiente:	$f(n) = n^2$
Bucle cúbico:	$f(n) = n^3$

EJERCICIOS

1.1. El siguiente algoritmo pretende calcular el cociente entero de dos enteros positivos (un dividendo y un divisor) contando el número de veces que el divisor se puede restar del dividendo antes de que se vuelva de menor valor que el divisor. Por ejemplo, 14/3 proporcionará el resultado 4 ya que 3 se puede restar de 14 cuatro veces. ¿Es correcto? Justifique su respuesta.

```
Cuenta ← 0;
Resto  ← Dividendo;
repetir
   Resto  ← Resto − Divisor
   Cuenta ← Cuenta + 1
hasta _ que (Resto < Divisor)
Cociente  ← Cuenta
```

1.2. El siguiente algoritmo está diseñado para calcular el producto de dos enteros negativos `x` e `y` por acumulación de la suma de copias de `y` (es decir, 4 por 5 se calcula acumulando la suma de cuatro cinco veces). ¿Es correcto? Justifique su respuesta.

```
producto ← y;
cuenta ← 1;
mientras (cuenta < x) hacer
   producto ← producto + y;
   cuenta ← cuenta + 1
fin _ mientras
```

1.3. Determinar la *O-grande* de los algoritmos escritos en los ejercicios 1.2 y 1.3.

1.4. Diseñar un algoritmo que calcule el número de veces que una cadena de caracteres aparece como una subcadena de otra cadena. Por ejemplo, `abc` aparece dos veces en la cadena `abcdabc`, y la cadena `aba` aparece dos veces en la cadena `ababa`.

1.5. Diseñar un algoritmo para determinar si un número *n* es primo. (Un número primo sólo puede ser divisible por él mismo y por la unidad.)

1.6. Determinar la *O-grande* de los algoritmos que resuelven los ejercicios 1.4 y 1.5.

1.7. Escribir un algoritmo que calcule la superficie de un triángulo en función de la base y la altura `(S = ½ Base x Altura)`.

1.8. Escribir un algoritmo que calcule y muestre la longitud de la circunferencia y el área de un círculo de radio dado.

1.9. Escribir un algoritmo que indique si una palabra leída del teclado es un palíndromo. Un *palíndromo* (capicúa) es una palabra que se lee igual en ambos sentidos como “radar”.

1.10. Calcular la eficiencia de los siguientes algoritmos:

a.
```
i = 1
mientras (i <= n)
   j = 1
   mientras (j <= n)
      j = j * 2
   fin _ mientras
   i = i + 1
fin _ mientras
```

b.
```
i = 1
mientras (i <= n)
   j = 1
   mientras (j <= i)
      j = j + 1
   fin _ mientras
   i = i + 1
fin _ mientras
```

c.
```
i = 1
mientras (i <= 10)
   j = 1
   mientras (j <= 10)
      j = j + 1
   fin _ mientras
   i = i + 2
fin _ mientras
```

CAPITULO 2

Tipos de datos: Clases y objetos

Objetivos

Con el estudio de este capítulo usted podrá:

- Definir lo que es un tipo de datos.
- Conocer los tipos de datos básicos.
- Conocer los tipos de datos estructurados.
- Especificar los tipos abstractos de datos.
- Entender el concepto de encapsulación de datos a través de las clases.
- Definir las clases como una estructura que encierra datos y métodos.
- Especificar tipos abstractos de datos a través de una clase.
- Establecer controles de acceso a los miembros de una clase.
- Identificar los miembros dato de una clase como la representación de las propiedades de un objeto.
- Identificar los métodos de una clase con el comportamiento o funcionalidad de los objetos.

Contenido

Conceptos clave

- ♦ Abstracción.
- ♦ Componentes.
- ♦ Constructores.
- ♦ Encapsulación.
- ♦ Especificadores de acceso: *public, protected, private.*
- ♦ Interfaz.
- ♦ Ocultación de la información.
- ♦ Reutilización.
- ♦ Software.
- ♦ Tipos de datos y variables.

Para profundizar (página web: www.mhe.es/joyanes)

- Aplicación del tipo abastracto de dato conjunto.

INTRODUCCIÓN

En este capítulo se examinan los conceptos de *modularidad* y *abstracción de datos*. La modularidad es la posibilidad de dividir una aplicación en piezas más pequeñas llamadas módulos. La *abstracción de datos* es la técnica para inventar nuevos tipos de datos que sean más adecuados a una aplicación y, por consiguiente, faciliten la escritura del programa. La técnica de abstracción de datos es una técnica potente de propósito general que, cuando se utiliza adecuadamente, puede producir programas más cortos, más legibles y flexibles.

Los lenguajes de programación soportan en sus compiladores *tipos de datos fundamentales o básicos (predefinidos)*, tales como `int`, `char` y `float` en Java, C y C++. Lenguajes de programación como Java tienen características que permiten ampliar el lenguaje añadiendo sus propios tipos de datos.

Un tipo de dato definido por el programador se denomina *tipo abstracto de dato, **TAD**, (abstract data type, **ADT**)*. El término abstracto se refiere al medio en que un programador abstrae algunos conceptos de programación creando un nuevo tipo de dato.

La modularización de un programa utiliza la noción de tipo abstracto de dato (TAD) siempre que sea posible. Si el lenguaje de programación soporta los tipos que desea el usuario y el conjunto de operaciones sobre cada tipo, se obtiene un nuevo tipo de dato denominado TAD.

Una **clase** es un tipo de dato que contiene código (métodos) y datos. Una clase permite encapsular todo el código y los datos necesarios para gestionar un tipo específico de un elemento de programa, como una ventana en la pantalla, un dispositivo conectado a una computadora, una figura de un programa de dibujo o una tarea realizada por una computadora. En este capítulo se aprenderá a crear (definir y especificar) y a utilizar clases individuales.

2.1. ABSTRACCIÓN EN LENGUAJES DE PROGRAMACIÓN

Los lenguajes de programación son las herramientas mediante las cuales los diseñadores de lenguajes pueden implementar los modelos abstractos. La abstracción ofrecida por los lenguajes de programación se puede dividir en dos categorías: *abstracción de datos* (perteneciente a los datos) y *abstracción de control* (perteneciente a las estructuras de control).

Desde comienzos de la década de los sesenta, cuando se desarrollaron los primeros lenguajes de programación de alto nivel, ha sido posible utilizar las abstracciones más primitivas de ambas categorías (variables, tipos de datos, procedimientos, control de bucles, etc.).

2.1.1. Abstracciones de control

Los microprocesadores ofrecen directamente sólo dos mecanismos para controlar el flujo y ejecución de las instrucciones: *secuencia* y *salto*. Los primeros lenguajes de programación de alto nivel introdujeron las estructuras de control: sentencias de bifurcación `(if)` y bucles (`for, while, do-loop,` etc.).

Las estructuras de control describen el orden en el que se ejecutan las sentencias o grupos de sentencia *(unidades de programa)*. Las unidades de programa se utilizan como bloques básicos de la clásica descomposición "descendente". En todos los casos, los subprogramas constituyen una herramienta potente de abstracción ya que, durante su implementación, el programador describe en detalle cómo funcionan. Cuando el subprograma se llama, basta con conocer lo que hace y no cómo lo hace. De este modo, se convierten en cajas negras que amplían el lenguaje de programación a utilizar. En general, los subprogramas son los mecanismos más ampliamente utilizados para reutilizar código, a través de colecciones de subprogramas en bibliotecas.

Las abstracciones y las estructuras de control se clasifican en estructuras de control a nivel de sentencia y a nivel de unidades. Las abstracciones de control a nivel de unidad se conocen como *abstracciones procedimentales*.

Abstracción *procedimental* (por procedimientos)

Es esencial para diseñar software modular y fiable la abstracción procedimental que se basa en la utilización de procedimientos o funciones sin preocuparse de cómo se implementan. Esto es posible sólo si conocemos qué hace el procedimiento; esto es, conocemos la sintaxis y la semántica que utiliza el procedimiento o función. La abstracción aparece en los subprogramas debido a las siguientes causas:

- Con el nombre de los subprogramas, un programador puede asignar una descripción abstracta que captura el significado global del subprograma. Utilizando el nombre en lugar de escribir el código, permite al programador aplicar la acción en términos de su descripción de alto nivel en lugar de sus detalles de bajo nivel.
- Los subprogramas proporcionan ocultación de la información. Las variables locales y cualquier otra definición local se encapsulan en el subprograma, ocultándose de forma que no pueden utilizarse fuera del subprograma. Por consiguiente, el programador no tiene que preocuparse sobre las definiciones locales.
- Los parámetros de los subprogramas, junto con la ocultación de la información anterior, permiten crear subprogramas que constituyen entidades de *software* propias. Los detalles locales de la implementación pueden estar ocultos, mientras que los parámetros se pueden utilizar para establecer la interfaz *pública*.

En Java, la abstracción procedimental se establece con los métodos o funciones miembros de clases.

Otros mecanismos de abstracción de control

La evolución de los lenguajes de programación ha permitido la aparición de otros mecanismos para la abstracción de control, como *manejo de excepciones, corrutinas, unidades concurrentes* o *plantillas (templates)*. Estas construcciones son soportadas por los lenguajes de programación basados y orientados a objetos, como Java, Modula-2, Ada, C++, Smalltalk o Eiffel.

2.1.2. Abstracciones de datos

Los primeros pasos hacia la abstracción de datos se crearon con lenguajes tales como FORTRAN, COBOL y ALGOL 60, con la introducción de tipos de variables diferentes, que manipulaban enteros, números reales, caracteres, valores lógicos, etc. Sin embargo, estos tipos de datos no podían ser modificados y no siempre se ajustaban al tipo de uno para el que se necesitaban. Por ejemplo, el tratamiento de cadenas es una deficiencia en FORTRAN, mientras que la precisión y fiabilidad para cálculos matemáticos es muy alta.

La siguiente generación de lenguajes, incluyendo Pascal, SIMULA-67 y ALGOL 68, ofreció una amplia selección de tipos de datos y permitió al programador modificar y ampliar los tipos de datos existentes mediante construcciones específicas (por ejemplo, *arrays* y registros). Además, SIMULA-67 fue el primer lenguaje que mezcló datos y procedimientos mediante la construcción de clases, que eventualmente se convirtió en la base del desarrollo de programación orientada a objetos.

La *abstracción de datos* es la técnica de programación que permite inventar o definir nuevos tipos de datos (tipos de datos definidos por el usuario) adecuados a la aplicación que se desea realizar. La abstracción de datos es una técnica muy potente que permite diseñar programas más cortos, legibles y flexibles. La esencia de la abstracción es similar a la utilización de un tipo de dato, cuyo uso se realiza sin tener en cuenta cómo está representado o implementado.

Los tipos de datos son abstracciones y el proceso de construir nuevos tipos se llama abstracción de datos. Los nuevos tipos de datos definidos por el usuario se llaman *tipos abstractos de datos*.

El concepto de tipo, tal como se definió en Pascal y ALGOL 68, ha constituido un hito importante para la realización de un lenguaje capaz de soportar programación estructurada. Sin embargo, estos lenguajes no soportan totalmente una metodología orientada a objetos. La abstracción de datos útil para este propósito no sólo clasifica objetos de acuerdo a su estructura de representación, sino que los clasifican de acuerdo al comportamiento esperado. Tal comportamiento es expresable en términos de operaciones que son significativas sobre esos datos, y las operaciones son el único medio para crear, modificar y acceder a los objetos.

En términos más precisos, Ghezzi indica que un tipo de dato definible por el usuario se denomina tipo abstracto de dato (TAD) si:

- Existe una construcción del lenguaje que le permite asociar la representación de los datos con las operaciones que lo manipulan;
- La representación del nuevo tipo de dato está oculta de las unidades de programa que lo utilizan [GHEZZI 87].

Las clases de Java o de C++ cumplen las dos condiciones: agrupan los datos junto a las operaciones, y su representación queda oculta de otras clases.

Los tipos abstractos de datos proporcionan un mecanismo adicional mediante el cual se realiza una separación clara entre la *interfaz* y la *implementación* del tipo de dato. La implementación de un tipo abstracto de dato consta de:

1. La representación: elección de las estructuras de datos.
2. Las operaciones: elección de los algoritmos.

La interfaz del tipo abstracto de dato se asocia con las operaciones y datos *visibles* al exterior del TAD.

2.2. TIPOS ABSTRACTOS DE DATOS

Algunos lenguajes de programación tienen características que nos permiten ampliar el lenguaje añadiendo sus propios tipos de datos. Un tipo de dato definido por el programador se denomina tipo abstracto de datos (**TAD**) para diferenciarlo del tipo fundamental (predefinido) de datos. Por ejemplo, en Java, el tipo `Punto`, que representa las coordenadas `x` e `y` de un sistema de coordenadas rectangulares, no existe. Sin embargo, es posible implementar el tipo abstracto de datos, considerando los valores que se almacenan en las variables y qué operaciones están disponibles para manipular estas variables. En esencia, un tipo abstracto es un tipo de dato que consta de datos (estructuras de datos propias) y operaciones que se pueden realizar sobre ellos. Un TAD se compone de *estructuras de datos* y los *procedimientos* o *funciones* que manipulan esas estructuras de datos.

> ***Para recordar***
>
> Un tipo abstracto de datos puede definirse mediante la ecuación:
>
> **TAD** = Representación (datos) + Operaciones (funciones y procedimientos)

La estructura de un tipo abstracto de dato *(clase)*, desde un punto de vista global, se compone de la interfaz y de la implementación (Figura 2.1).

Las estructuras de datos reales elegidas para almacenar la representación de un tipo abstracto de datos son invisibles a los usuarios o clientes. Los algoritmos utilizados para implementar cada una de las operaciones de los TAD están encapsuladas dentro de los propios TAD. La característica de ocultamiento de la información significa que los objetos tienen *interfaces públicas*. Sin embargo, las representaciones e implementaciones de esas interfaces son *privadas*.

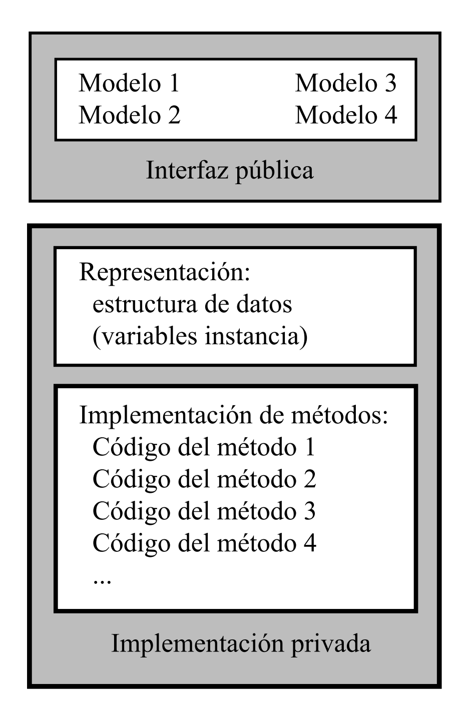


Figura 2.1 Estructura de un tipo abstracto de datos (TAD)

2.2.1. Ventajas de los tipos abstractos de datos

Un tipo abstracto de datos es un modelo (estructura) con un número de operaciones que afectan a ese modelo. Los tipos abstractos de datos proporcionan numerosos beneficios al programador, que se pueden resumir en los siguientes:

1. Permiten una mejor conceptualización y modelización del mundo real. Mejoran la representación y la comprensibilidad. Clarifican los objetos basados en estructuras y comportamientos comunes.

2. Mejoran la robustez del sistema. Si hay características subyacentes en los lenguajes, permiten la especificación del tipo de cada variable. Los tipos abstractos de datos permiten la comprobación de tipos para evitar errores de tipo en tiempo de ejecución.
3. Mejoran el rendimiento (prestaciones). Para sistemas *tipeados* (tipificados), el conocimiento de los objetos permite la optimización de tiempo de compilación.
4. Separan la implementación de la especificación. Permiten la modificación y la mejora de la implementación sin afectar la interfaz pública del tipo abstracto de dato.
5. Permiten la extensibilidad del sistema. Los componentes de *software* reutilizables son más fáciles de crear y mantener.
6. Recogen mejor la semántica del tipo. Los tipos abstractos de datos agrupan o localizan las operaciones y la representación de atributos.

Un programa que maneja un TAD lo hace teniendo en cuenta las operaciones o la funcionalidad que tiene, sin interesarse por la representación física de los datos. Es decir, los *usuarios* de un TAD se comunican con éste a partir de la interfaz que ofrece el TAD mediante funciones de acceso. Podría cambiarse la implementación del tipo de dato sin afectar al programa que usa el TAD ya que para el programa está *oculta*.

2.2.2. Implementación de los TAD

Las unidades de programación de lenguajes que pueden implementar un TAD reciben distintos nombres:

Modula-2	*módulo*
Ada	*paquete*
C++	*clase*
Java	*clase*

En estos lenguajes se definen la *especificación* del TAD, que declara las operaciones y los datos, y la *implementación,* que muestra el código fuente de las operaciones, que permanece oculto al exterior del módulo.

2.3. ESPECIFICACIÓN DE LOS TAD

El objetivo de la especificación es describir el comportamiento del TAD; consta de dos partes, la descripción matemática del conjunto de datos y la de las operaciones definidas en ciertos elementos de ese conjunto de datos.

La especificación del TAD puede tener un enfoque *informal*, que describe los datos y las operaciones relacionadas en *lenguaje natural*. Otro enfoque mas riguroso, la *especificación formal*, supone suministrar un conjunto de *axiomas* que describen las operaciones en su aspecto *sintáctico* y *semántico*.

2.3.1. Especificación informal de un TAD

Consta de dos partes:

- Detallar en los datos del tipo los valores que pueden tomar.
- Describir las operaciones relacionándolas con los datos.

El formato que generalmente se emplea, primero especifica el nombre del TAD y los datos:

TAD *nombre del tipo* (valores y su descripción)

A continuación cada una de las operaciones con sus argumentos, y una descripción funcional en lenguaje natural, con este formato:

```
Operación(argumentos)
Descripción funcional
```

Como ejemplo, se va a especificar el tipo abstracto de datos *Conjunto:*

TAD Conjunto(colección de elementos sin duplicidades, pueden estar en cualquier orden, se usa para representar los conjuntos matemáticos con sus operaciones).

Operaciones, se ponen las operaciones básicas sobre conjuntos:

Conjuntovacio.
Crea un conjunto sin elementos.

Añadir(Conjunto, elemento).
Comprueba si el elemento forma parte del conjunto; en caso negativo, es añadido. La operación modifica al conjunto.

Retirar(Conjunto, elemento).
Si el elemento pertenece al conjunto, es eliminado de éste. La operación modifica al conjunto.

Pertenece(Conjunto, elemento).
Verifica si el elemento forma parte del conjunto, en cuyo caso devuelve *cierto*.

Esvacio(Conjunto).
Verifica si el conjunto no tiene elementos, en cuyo caso devuelve *cierto*.

Cardinal(Conjunto).
Devuelve el número de elementos del conjunto.

Union(Conjunto, Conjunto).
Realiza la operación matemática de la unión de dos conjuntos. La operación devuelve un conjunto con los elementos comunes y no comunes a los dos conjuntos.

Se pueden especificar más operaciones sobre conjuntos, todo dependerá de la aplicación que se quiera dar al *TAD*.

A tener en cuenta

La especificación informal de un TAD tiene como objetivo describir los datos del tipo y las operaciones según la funcionalidad que tienen. No sigue normas rígidas al hacer la especificación, simplemente indica, de forma comprensible, la acción que realiza cada operación.

2.3.2. Especificación formal de un TAD

La especificación formal proporciona un conjunto de axiomas que describen el comportamiento de todas las operaciones. La descripción ha de incluir una parte de sintaxis, en cuanto a los tipos

de los argumentos y al tipo del resultado, y una parte de semántica, donde se detalla la expresión del resultado que se obtiene para unos valores particulares de los argumentos. La especificación formal ha de ser lo bastante *potente* para que cumpla el objetivo de verificar la corrección de la implementación del TAD.

El esquema que sigue consta de una cabecera con el nombre del TAD y los datos:

```
TAD nombre del tipo (valores que toma los datos del tipo)
```

Le sigue la sintaxis de las operaciones, que lista las operaciones mostrando los tipos de los argumentos y el tipo del resultado:

Sintaxis

```
Operación(Tipo argumento, ...)   ->   Tipo resultado
```

A continuación se explica la semántica de las operaciones. Ésta se construye dando unos valores particulares a los argumentos de las operaciones, a partir de los que se obtiene una expresión resultado. Éste puede tener referencias a tipos ya definidos, valores de tipo lógico o referencias a otras operaciones del propio TAD.

Semántica

```
Operación(valores particulares argumentos) ⇒ expresión resultado
```

Al hacer una especificación formal, siempre hay operaciones definidas por sí mismas que se consideran *constructores* del TAD. Se puede decir que mediante estos constructores se generan todos los posibles valores del TAD. Normalmente, se elige como constructor la operación que inicializa (por ejemplo, *Conjuntovacío* en el *TAD Conjunto*) y la operación que añade un dato o elemento (esta operación es común a la mayoría de los tipos abstractos de datos). Se acostumbra a marcar con un asterisco las operaciones que son constructores.

A continuación se especifica formalmente el *TAD Conjunto*; para formar la expresión resultado se hace uso, si es necesario, de la sentencia alternativa *si-entonces-sino.*

TAD Conjunto(*colección de elementos sin duplicidades, pueden estar en cualquier orden, se usa para representar los conjuntos matemáticos con sus operaciones*).

Sintaxis

```
*Conjuntovacio                   ->   Conjunto
*Añadir(Conjunto, Elemento)      ->   Conjunto
Retirar(Conjunto, Elemento)      ->   Conjunto
Pertenece(Conjunto, Elemento)    ->   boolean
Esvacio(Conjunto)                ->   boolean
Cardinal(Conjunto)               ->   entero
Union(Conjunto, Conjunto)        ->   Conjunto
```

```
Semántica    ∀ e1,e2 ∈ Elemento y ∀ C,D ∈ Conjunto

   Añadir(Añadir(C, e1), e1)      ⇒ Añadir(C, e1)
   Añadir(Añadir(C, e1), e2)      ⇒ Añadir(Añadir(C, e2), e1)
   Retirar(Conjuntovacio, e1)     ⇒ Conjuntovacio
   Retirar(Añadir(C, e1), e2)     ⇒ si e1 = e2 entonces Retirar(C,e2)
                                    sino Añadir(Retirar(C,e2),e1)
```

```
Pertenece(Conjuntovacio, e1)  ⇒ falso
Pertenece(Añadir(C, e2), e1)  ⇒ si e1 = e2 entonces cierto
                                sino Pertenece(C, e1)
Esvacio(Conjuntovacio)        ⇒ cierto
Esvacio(Añadir(C, e1))        ⇒ falso
Cardinal(Conjuntovacio)       ⇒ Cero
Cardinal(Añadir(C, e1))       ⇒ si Pertenece(C,e1) entonces
                                     Cardinal(C)
                                sino 1 + Cardinal(C)
Union(Conjuntovacio,
            Conjuntovacio)    ⇒ Conjuntovacio
Union(Conjuntovacio,
            Añadir(C, e1))    ⇒ Añadir(C, e1)
Union(Añadir(C, e1), D)       ⇒ Añadir(Union(C, D), e1)
```

2.4. CLASES Y OBJETOS

El paradigma orientado a objetos nació en 1969 de la mano del doctor noruego Kristin Nygaard, que al intentar escribir un programa de computadora que describiera el movimiento de los barcos a través de un fiordo, descubrió que era muy difícil simular las mareas, los movimientos de los barcos y las formas de la línea de la costa con los métodos de programación existentes en ese momento. Descubrió que los elementos del entorno que trataba de modelar —barcos, mareas y línea de la costa de los fiordos— y las acciones que cada elemento podía ejecutar mantenían unas relaciones que eran más fáciles de manejar.

Las tecnologías orientadas a objetos han evolucionado mucho, pero mantienen la razón de ser del paradigma: combinación de la descripción de los elementos en un entorno de proceso de datos con las acciones ejecutadas por esos elementos. Las clases y los objetos, como instancias o ejemplares de ellas, son los elementos clave sobre los que se articula la orientación a objetos.

2.4.1. ¿Qué son objetos?

En el mundo real, las personas identifican los objetos como cosas que pueden ser percibidas por los cinco sentidos. Los objetos tienen propiedades específicas, como posición, tamaño, color, forma, textura, etc. que definen su estado. Los objetos también poseen ciertos comportamientos que los hacen diferentes de otros objetos.

Booch[1] define un *objeto* como "algo que tiene un estado, un comportamiento y una identidad". Imaginemos una máquina de una fábrica. El *estado* de la *máquina* puede estar *funcionando/parando* ("on/off"), hay que tener en cueta su potencia, velocidad máxima, velocidad actual, temperatura, etc. Su *comportamiento* puede incluir acciones para arrancar y parar la máquina, obtener su temperatura, activar o desactivar otras máquinas, conocer las condiciones de señal de error o cambiar la velocidad. Su *identidad* se basa en el hecho de que cada instancia de una máquina es única, tal vez identificada por un número de serie. Las características que se eligen para enfatizar el estado y el comportamiento se apoyarán en cómo un objeto máquina se utilizará en una aplicación. En un diseño de un programa orientado a objetos, se crea una abstracción (un modelo simplificado) de la máquina basada en las propiedades y en el comportamiento que son útiles en el tiempo.

---

[1] Booch, Grady. *Análisis y diseño orientado a objetos con aplicaciones*. Madrid: Díaz de Santos/Addison-Wesley, 1995 (libro traducido al español por los profesores Cueva y Joyanes).

Martin y Odell definen un objeto como "cualquier cosa, real o abstracta, en la que se almacenan datos y aquellos métodos (operaciones) que manipulan los datos". Para realizar esa actividad, se añaden a cada objeto de la clase los propios datos asociados con sus propios métodos miembro que pertenecen a la clase.

Un *mensaje* es una instrucción que se envía a un objeto y que, cuando se recibe, ejecuta sus acciones. Un mensaje incluye un identificador que contiene la acción que ha de ejecutar el objeto junto con los datos que necesita el objeto para realizar su trabajo. Los mensajes, por consiguiente, forman una ventana del objeto al mundo exterior.

El usuario de un objeto se comunica con el objeto mediante su *interfaz,* un conjunto de operaciones definidas por la clase del objeto de modo que sean todas visibles al programa. Una interfaz se puede considerar como una vista simplificada de un objeto. Por ejemplo, un dispositivo electrónico como una máquina de fax tiene una interfaz de usuario bien definida; por ejemplo, esa interfaz incluye el mecanismo de avance del papel, botones de marcado, el receptor y el botón "enviar". El usuario no tiene que conocer cómo está construida la máquina internamente, el protocolo de comunicaciones u otros detalles. De hecho, la apertura de la máquina durante el periodo de garantía puede anularla.

2.4.2. ¿Qué son clases?

En términos prácticos, una *clase* es un tipo definido por el usuario. Las clases son los bloques de construcción fundamentales de los programas orientados a objetos. Booch define una clase como "un conjunto de objetos que comparten una estructura y un comportamiento comunes".

Una clase contiene la especificación de los datos que describen un objeto junto con la descripción de las acciones que un objeto conoce cómo ha de ejecutar. Estas acciones se conocen como *servicios o métodos.* Una clase incluye también todos los datos necesarios para describir los objetos creados a partir de la clase. Estos datos se conocen como *atributos, variables o variables instancia*. El término *atributo* se utiliza en análisis y diseño orientado a objetos, y el término *variable instancia* se suele utilizar en programas orientados a objetos.

2.5. DECLARACIÓN DE UNA CLASE

Antes de que un programa pueda crear objetos de cualquier clase, ésta debe ser *definida*. La definición de una clase significa que se debe dar a la misma un nombre, dar nombre también a los elementos que almacenan sus datos y describir los métodos que realizarán las acciones consideradas en los objetos.

Las *definiciones* o *especificaciones* no son un código de programa ejecutable. Se utilizan para asignar almacenamiento a los valores de los atributos usados por el programa y reconocer los métodos que utilizará el programa. Normalmente, se sitúan en archivos formando los denominados *packages,* se utiliza un archivo para varias clases que están relacionadas.

Formato

```
class NombreClase
{
   lista_de_miembros
}
```

`NombreClase`	Nombre definido por el usuario que identifica la clase (puede incluir letras, números y subrayados).
`lista _ de _ miembros`	*métodos* y datos miembros de la clase.

Ejemplo 2.1

Definición de una clase llamada `Punto` *que contiene las coordenadas x e y de un punto en un plano.*

```
class Punto
{
   private int x;     // coordenada x
   private int y;     // coordenada y
   public Punto(int x _ , int y _ )  // constructor
   {
      x = x _ ;
      y = y _ ;
   }
   public Punto()  // constructor sin argumentos
   {
      x = y = 0;
   }
   public  int leerX()   // devuelve el valor de x
   {
      return x;
   }
   public  int leerY()   // devuelve el valor de y
   {
      return y;
   }
   void fijarX(int valorX)   // establece el valor de x
   {
      x = valorX;
   }
   void fijarY(int valorY)   // establece el valor de y
   {
      y = valorY;
   }
}
```

Ejemplo 2.2

Declaración de la clase `Edad`.

```
class Edad
{
    private int edadHijo, edadMadre, edadPadre; ————→ datos

    public Edad(){...}      ————→  método especial: constructor

    public void iniciar(int h,int e,int p){...}————→ métodos
    public int leerHijo(){...}
```

```
    public int leerMadre(){...}
    public int leerPadre(){...}
}
```

2.5.1. Objetos

Una vez que una clase ha sido definida, un programa puede contener una *instancia* de la clase, denominada *objeto de la clase*. Un objeto se crea con el operador `new` aplicado a un constructor de la clase. Un objeto de la clase `Punto` inicializado a las coordenadas `(2,1)` sería:

```
new Punto(2,1);
```

El operador `new` crea el objeto y devuelve una referencia al objeto creado. Esta referencia se asigna a una variable del tipo de la clase. El objeto permanecerá vivo siempre que esté referenciado por una variable de la clase que es instancia.

Formato para definir una referencia

```
NombreClase   varReferencia;
```

Formato para crear un objeto

```
varReferencia = new NombreClase(argmntos _ constructor);
```

Toda clase tiene uno o mas métodos, denominados constructores, para inicializar el objeto cuando es creado; tienen el mismo nombre que el de la clase, no tienen tipo de retorno y pueden estar sobrecargados. En la clase `Edad` del Ejemplo 2.2, el constructor no tiene argumentos, se puede crear un objeto:

```
Edad f = new Edad();
```

El *operador de acceso* a un miembro (.) selecciona un miembro individual de un objeto de la clase. Por ejemplo:

```
Punto p;
p = new Punto();
p.fijarX (100);
System.out.println(" Coordenada  x es " + p.leerX());
```

El operador punto (.) se utiliza con los nombres de los métodos y variables instancia para especificar que son miembros de un objeto.

Ejemplo: `Clase DiaSemana,` contiene el método `visualizar()`

```
DiaSemana hoy;          // hoy es una referencia
hoy = new DiaSemana();  // se ha creado un objeto
hoy.visualizar();       // llama al método visualizar()
```

2.5.2. Visibilidad de los miembros de la clase

Un principio fundamental en programación orientada a objetos es la *ocultación de la información,* que significa que a determinados datos del interior de una clase no se puede acceder por métodos externos a ella. El mecanismo principal para ocultar datos es ponerlos en una clase y hacerlos *privados*. A los datos o métodos privados sólo se puede acceder desde dentro de la clase. Por el contrario, los datos o métodos públicos son accesibles desde el exterior de la clase.

Figura 2.2 Secciones pública y privada de una clase

Para controlar el acceso a los miembros de la clase se utilizan tres *especificadores de acceso:* `public`, `private` y `protected`. Cada miembro de la clase está precedido del especificador de acceso que le corresponde.

Formato

```
class NombreClase
{
   private declaración miembro privado; // miembros privados

   protected declaración miembro protegido;  // miembros protegidos

   public declaración miembro público;  // miembros públicos
}
```

El especificador `public` define miembros públicos, que son aquellos a los que se puede acceder por cualquier método desde fuera de la clase. A los miembros que siguen al especificador `private` sólo se puede acceder por métodos miembros de la misma clase. A los miembros que siguen al especificador `protected` se puede acceder por métodos miembro de la misma clase o de clases derivadas, así como por métodos de otras clases que se encuentran en el mismo *paquete*. Los especificadores `public`, `protected` y `private` pueden aparecer en cualquier orden. Si no se especifica acceso (*acceso por defecto*) a un miembro de una clase, a éste se puede acceder desde los métodos de la clase y desde cualquier método de las clases del *paquete* en el que se encuentra.

Ejemplo 2.3

Declaración de la clase `Foto` *y* `Marco` *con miembros declarados con distinta visibilidad. Ambas clases forman parte del paquete* `soporte`.

```
package soporte;

class Foto
{
   private int nt;
   private char opd;

   String q;
   public Foto(String r)  // constructor
   {
      nt = 0;
      opd = 'S';
      q = new String(r);
   }
   public double mtd(){...}
}

class Marco
{
   private double p;
   String t;
   public Marco() {...}

   public void poner()
   {
      Foto u = new Foto("Paloma");
      p = u.mtd();
      t = "**" + u.q + "**";
   }
}
```

Tabla 2.1 Visibilidad, "x" indica que el acceso está permitido

Tipo de miembro	Miembro de la misma clase	Miembro de una clase derivada	Miembro de clase del paquete	Miembro de clase de otro paquete
`Private`	x			
En blanco	x		x	
`Protected`	x	x	x	
`Public`	x	x	x	x

Aunque las especificaciones *públicas, privadas* y *protegidas* pueden aparecer en cualquier orden, en Java los programadores suelen seguir ciertas reglas en el diseño que citamos a continuación, para que usted pueda elegir la que considere más eficiente.

1. Poner los miembros privados primero, debido a que contienen los atributos (datos).
2. Colocar los miembros públicos primero, debido a que los métodos y los constructores son la interfaz del usuario de la clase.

En realidad, tal vez el uso más importante de los especificadores de acceso es implementar la ocultación de la información. El *principio de ocultación* de la información indica que toda la interacción con un objeto se debe restringir a utilizar una interfaz bien definida que permita ignorar los detalles de implementación de los objetos. Por consiguiente, los datos y los métodos

públicos forman la interfaz externa del objeto, mientras que los elementos *privados* son los aspectos internos que no necesitan ser accesibles para usar el objeto. Los elementos de una clase sin especificador y los `protected` tienen las mismas propiedades que los públicos respecto a las clases del paquete.

> El principio de *encapsulamiento* significa que las estructuras de datos internas utilizadas en la implementación de una clase no pueden ser accesibles directamente al usuario de la clase.

2.5.3. Métodos de una clase

Las métodos en Java siempre son miembros de clases, no hay métodos o funciones fuera de las clases. La implementación de los métodos se incluye dentro del cuerpo de la clase. La Figura 2.3 muestra la declaración completa de una clase.

```
class Producto        <---------  nombre de la clase
{                     <---------  Acceso para almacenamiento de datos
   private int numProd;           Declaraciones para almacenamiento
   private String nomProd;  <---  de datos
   private String descripProd;
   private double precioProd;
   private int numUnidades;

   public  Producto() {...} <---------- métodos
   public  Producto(int n,char[]nom,char[]des,double p,int nu){   }
   public  void verProducto(){...}
   public  double obtenerPrecio(){...}
   public  void actualizarProd(int b) {...}
}
```

Figura 2.3 Definición típica de una clase

Ejemplo 2.4

La clase Racional *define el numerador y el denominador característicos de un número racional. Por cada dato se proporciona un método miembro que devuelve su valor y un método que asigna numerador y denominador. Tiene un constructor que inicializa un objeto a* 0/1.

```
class Racional
{
   private int numerador;
   private int denominador;

   public Racional()
   {
      numerador = 0;
      denominador = 1;
   }
```

```
    public int leerN() { return numerador; }
    public int leerD() { return denominador; }
    public  void fijar (int n, int d)
    {
      numerador = n;
      denominador = d;
    }
  }
```

Ejercicio 2.1

Definir una clase `DiaAnyo` *que contenga los atributos mes y día, el método* `igual()` *y el método* `visualizar()`*. El mes se registra como un valor entero (1, Enero; 2, Febrero; etc.). El día del mes se registra en la variable entera* `día`*. Escribir un programa que compruebe si una fecha es la de su cumpleaños.*

El método `main()` de la clase principal, `Cumple`, crea un objeto `DiaAnyo` y llama al método `igual()` para determinar si coincide la fecha del objeto con la fecha de su cumpleaños, que se ha leído del dispositivo de entrada.

```
  import java.io.*;

  class DiaAnyo
  {
    private int mes;
    private int dia;

    public DiaAnyo(int d, int m)
    {
      dia = d;
      mes = m;
    }
    public boolean igual(DiaAnyo d)
    {
      if ((dia == d.dia) && (mes == d.mes))
        return true;
      else
        return false;
    }
    // muestra en pantalla el mes y el día
    public void visualizar()
    {
      System.out.println("mes = " + mes + " , dia = " + dia);
    }
  }
    // clase principal, con método main
  public class Cumple
  {
    public static void main(String[] ar)throws IOException
    {
      DiaAnyo hoy;
      DiaAnyo cumpleanyos;
      int d, m;
      BufferedReader entrada = new BufferedReader(
                           new InputStreamReader(System.in));
```

```
      System.out.print("Introduzca fecha de hoy, dia: ");
      d = Integer.parseInt(entrada.readLine());
      System.out.print("Introduzca el número de mes: ");
      m = Integer.parseInt(entrada.readLine());

      hoy = new DiaAnyo(d,m);

      System.out.print("Introduzca su fecha de nacimiento, dia: ");
      d = Integer.parseInt(entrada.readLine());
      System.out.print("Introduzca el número de mes: ");
      m = Integer.parseInt(entrada.readLine());
      cumpleanyos = new DiaAnyo(d,m);

      System.out.print( " La fecha de hoy es ");
      hoy.visualizar();
      System.out.print( " Su fecha de nacimiento es ");
      cumpleanyos.visualizar();

      if (hoy.igual(cumpleanyos))
         System.out.println( "¡Feliz cumpleaños ! ");
      else
         System.out.println( "¡Feliz dia ! ");
   }
}
```

2.5.4. Implementación de las clases

El código fuente de la definición de una clase, con todos sus métodos y variables instancia se almacena en archivos de texto con extensión `.java` y el nombre de la clase, por ejemplo, `Racional.java`. Normalmente, se sitúa la implementación de cada clase en un archivo independiente.

Las clases pueden proceder de diferentes fuentes:

- Se pueden declarar e implementar sus propias clases. El código fuente siempre estará disponible; pueden estar organizadas por paquetes.
- Se pueden utilizar clases que hayan sido escritas por otras personas o incluso que se hayan comprado. En este caso, se puede disponer del código fuente o estar limitado a utilizar el *bytecode* de la implementación. Será necesario disponer del paquete donde se encuentran.
- Se pueden utilizar clases de los diversos *packages* que acompañan a su software de desarrollo Java.

2.5.5. Clases públicas

La declaración de una clase puede incluir el modificador `public` como prefijo en la cabecera de la clase. Por ejemplo:

```
public class Examen
{
   // miembros de la clase
}
```

La clase `Examen` puede ser utilizada por las clases que se encuentran en su mismo archivo *(package)*, o por clases de cualquier otro paquete o archivo. Habitualmente, las clases se definen

como `public`, a no ser que se quiera restringir su uso a las clases del paquete. Una clase declarada sin el prefijo `public` establece una restricción importante, y es que sólo podrá ser utilizada por las clases definidas en el mismo paquete.

> ***Advertencia***
>
> El especificador de acceso `public` es el único que se puede especificar en la cabecera de una clase.

2.6. PAQUETES

Los paquetes son la forma que tiene Java de organizar los archivos con las clases necesarias para construir las aplicaciones. Java incorpora varios paquetes, por ejemplo `java.lang` o `java.io`, con las clases básicas para construir programas: `System, String, Integer ...`

2.6.1. Sentencia package

¿Cómo se puede definir un paquete? La sentencia `package` se utiliza para este cometido. En primer lugar se debe incluir la sentencia `package` como primera línea del archivo fuente de cada una de las clases del paquete. Por ejemplo, si las clases `Lapiz`, `Boligrafo` y `Folio` se van a organizar formando el paquete `escritorio`, el esquema a seguir es el siguiente:

```
// archivo fuente Lapiz.java

package escritorio;

public class Lapiz
{

   // miembros de clase Lapiz

}
// archivo fuente Boligrafo.java

package escritorio;

public class Boligrafo
{

   // miembros de clase Boligrafo

}

// archivo fuente Folio.java

package escritorio;

public class Folio
{

   // miembros de clase Folio

}
```

Formato

```
package NombrePaquete;
```

En segundo lugar, una vez creado el archivo fuente de cada clase del paquete, se deben ubicar cada uno en un subdirectorio con el mismo nombre que el del paquete. En el esquema anterior se ubicarán los archivos `Lapiz.java`, `Boligrafo.java` y `Folio.java` en el camino `escritorio`.

El uso de paquetes tiene dos beneficios importantes:

1. Las restricciones de visibilidad son menores entre clases que están dentro del mismo paquete. Desde cualquier clase de un paquete, los miembros `protected` y los miembros sin modificador de visibilidad son accesibles, pero no lo son desde clases de otros paquetes.
2. La selección de las clases de un paquete se puede abreviar con la sentencia `import` del paquete.

2.6.2. Import

Las clases que se encuentran en los paquetes se identifican utilizando el nombre del paquete, el selector punto (.) y, a continuación, el nombre de la clase. Por ejemplo, la declaración de la clase `Arte` con atributos de la clase `PrintStream` (paquete `java.io`) y `Lapiz` (paquete escritorio):

```
public class Arte
{
   private java.io.PrintStream salida;
   private escritorio.Lapiz p;
}
```

La sentencia `import` facilita la selección de una clase, permite escribir únicamente su nombre, evitando el nombre del paquete. La declaración anterior se puede abreviar:

```
import java.io.PrintStream;
import escritorio.*;
public class Arte
{
   private PrintStream salida;
   private Lapiz p;
}
```

La sentencia `import` debe aparecer antes de la declaración de las clases, a continuación de la sentencia `package`. Tiene dos formatos:

Formato

```
import identificadorpaquete.nombreClase;

import identificadorpaquete.*;
```

El primer formato especifica una clase concreta. El segundo formato indica que para todas las clases de un paquete no hace falta cualificar el nombre de la clase con el nombre del paquete.

Con frecuencia se utiliza el formato `.*`. Tiene la ventaja de poder simplificar cualquier clase del paquete, pero se pueden señalar los siguientes problemas:

- Se desconoce qué clases concretas del paquete se están utilizando. Por contra, con una sentencia `import` por cada clase utilizada se conocen las clases empleadas.
- Puede haber colisiones entre nombres de clases declaradas en el archivo y nombres de clases del paquete.
- Mayor tiempo de compilación, debido a que el compilador busca la existencia de cualquier clase en el paquete.

Nota

Aunque aparezca la sentencia `import paquete.*`, el compilador genera *bytecode* sólo para las clases utilizadas.

Ejemplo 2.5

Se crea el paquete `numerico` *con la clase* `Random` *y se utiliza la clase en una aplicación.*

```
package numerico;

public Random
{
   // ...
}
```

Al utilizar la clase en otro archivo:

```
import java.util.*
import numerico.*;
```

En el paquete `java.util` se encuentra la clase `Random`, por ello se produce una ambigüedad con la clase `Random` del paquete `numerico`. Es necesario cualificar completamente el nombre de la clase `Random` de, por ejemplo, `java.util`.

```
java.util.Random aleatorio; // define una referencia
```

2.7. CONSTRUCTORES

Un *constructor* es un método que se ejecuta automáticamente cuando se crea un objeto de una clase. Sirve para inicializar los miembros de la clase.

El constructor tiene el mismo nombre que la clase. Cuando se define no se puede especificar un valor de retorno, nunca devuelve un valor. Sin embargo, puede tomar cualquier número de argumentos.

Reglas

1. El constructor tiene el mismo nombre que la clase.
2. Puede tener cero o más argumentos.
3. No tiene tipo de retorno.

Ejemplo 2.6

La clase `Rectangulo` *tiene un constructor con cuatro parámetros.*

```
public class Rectangulo
{
   private int izdo;
   private int superior;
   private int dcha;
   private int inferior;
   // constructor
   public Rectangulo(int iz, int sr, int d, int inf)
   {
      izdo = iz;
      superior = sr;
      dcha = d;
      inferior = inf;
   }
   // definiciones de otros métodos miembro
}
```

Al crear un objeto, se pasan los valores de los argumentos al constructor con la misma sintaxis que la de llamada a un método. Por ejemplo:

```
Rectangulo Rect = new Rectangulo(25, 25, 75, 75);
```

Se ha creado una instancia de `Rectangulo` pasando valores concretos al constructor de la clase, y de esta forma queda inicializado.

2.7.1. Constructor por defecto

Un constructor que no tiene parámetros se llama *constructor por defecto*. Un constructor por defecto normalmente inicializa los miembros dato de la clase con valores por defecto.

> ***Regla***
>
> Java crea automáticamente un constructor por defecto cuando no existen otros constructores. Tal constructor inicializa las variables de tipo numérico `(int, float ...)` a cero, las variables de tipo `boolean` a `true` y las referencias a `null`.

Ejemplo 2.7

El constructor por defecto inicializa `x` *e* `y` *a* `0`.

```
public class Punto
{
   private int x;
   private int y;

   public Punto()     // constructor por defecto
   {
      x = 0;
```

```
        y = 0;
    }
}
```

Cuando se crea un objeto `Punto`, sus miembros dato se inicializan a `0`.

```
Punto P1 = new Punto() ;        // P1.x == 0, P1.y == 0
```

Precaución

Tenga cuidado con la escritura de una clase con sólo un constructor con argumentos. Si se omite un constructor sin argumento, no será posible utilizar el constructor por defecto. La definición `NomClase c = new NomClase()` no será posible.

2.7.2. Constructores sobrecargados

Al igual que se puede sobrecargar un método de una clase, también se puede sobrecargar el constructor de una clase. De hecho, los constructores sobrecargados son bastante frecuentes y proporcionan diferentes alternativas para inicializar objetos.

Regla

Para prevenir a los usuarios de la clase de crear un objeto sin parámetros, se puede: (1) omitir el constructor por defecto, o bien (2) hacer el constructor privado.

Ejemplo 2.8

La clase `EquipoSonido` *se define con tres constructores: uno por defecto, otro con un argumento de tipo cadena y el tercero con tres argumentos.*

```
public class EquipoSonido
{
   private int potencia;
   private int voltios;
   private int numCd;
   private String marca;

   public  EquipoSonido()    // constructor por defecto
   {
      marca = "Sin marca";
      System.out.println("Constructor por defecto");
   }
   public EquipoSonido(String mt)
   {
      marca = mt;
      System.out.println("Constructor con argmto cadena ");
   }
   public EquipoSonido(String mt, int p, int v)
   {
      marca = mt;
```

```
      potencia = p;
      voltios = v;
      numCd = 20;
      System.out.println("Constructor con tres argumentos ");
   }
   public double factura(){...}

};
```

La instanciación de un objeto `EquipoSonido` puede hacerse llamando a cualquier constructor:

```
EquipoSonido rt, gt, ht;        // define tres referencias
rt = new EquipoSonido();        // llamada al constructor por defecto
gt = new EquipoSonido("JULAT");
rt = new EquipoSonido("PARTOLA",35,220);
```

2.8. RECOLECCIÓN DE OBJETOS

En Java, un objeto siempre ha de estar referenciado por una variable; en el momento en que un objeto deja de estar referenciado, se activa la rutina de recolección de memoria, se puede decir que el objeto es liberado y la memoria que ocupa puede ser reutilizada. Por ejemplo:

```
Punto p = new Punto(1,2);
```

la sentencia `p = null` provoca que el objeto `Punto` sea liberado automáticamente.

El propio sistema se encarga de recolectar los objetos en desuso para aprovechar la memoria ocupada. Para ello, hay un proceso que se activa periódicamente y toma los objetos que no están referenciados por ninguna variable. El proceso lo realiza el método `System.gc` *(garbage collection)*. Por ejemplo, el siguiente método crea objetos `Contador` que se *liberan* al perder su referencia.

```
void objetos()
{
   Contador k, g, r, s;
   // se crean cuatro objetos
   k = new Contador();
   g = new Contador();
   r = new Contador();
   s = new Contador();
   /* la siguiente asignación hace que g referencie al mismo
        objeto que k, además el objeto original de g será
        automáticamente recolectado. */
   g = k;
   /* ahora no se activa el recolector porque g sigue apuntando al
      objeto. */
   k = null;
   /* a continuación sí se activa el recolector para el objeto
       original de r. */
   r = new Contador();

}  // se liberan los objetos actuales apuntados por g, r, s
```

2.8.1. Método `finalize()`

El método `finalize()` es especial, se llama automáticamente si ha sido definido en la clase, justo antes que la memoria del objeto *recolectado* vaya a ser devuelta al sistema. El método no es un destructor del objeto, no libera memoria; en algunas aplicaciones, se puede utilizar para liberar ciertos recursos del sistema.

REGLA

`finalize()` es un método especial con estas características:

- No devuelve valor, es de tipo `void`.
- No tiene argumentos.
- No puede sobrecargarse.
- Su visibilidad es `protected`.

Ejercicio 2.2

Se declaran dos clases, cada una con su método `finalize()`. *El método* `main()` *crea objetos de ambas clases; las variables que referencian a los objetos se modifican para que cuando se active la recolección automática de objetos se libere la memoria de estos; hay una llamada a* `System.gc()` *para no esperar a la llamada interna del sistema.*

```
class Demo
{
   private int datos;
   public  Demo(){datos = 0;}
   protected void finalize()
   {
      System.out.println("Fin de objeto Demo");
   }
}

class Prueba
{
   private  double x;
   public  Prueba (){x = -1.0;}
   protected void finalize()
   {
      System.out.println("Fin de objeto Prueba");
   }
}

public class ProbarDemo
{
   public static void main(String[] ar)
   {
      Demo d1, d2;
      Prueba p1, p2;
      d1 = new Demo();
      p1 = new Prueba();
      System.gc();  // no se libera ningún objeto
```

```
      p2 = p1;
      p1 = new Prueba();
      System.gc();  // no se libera ningún objeto
      p1 = null;
      d1 = new Demo();
      System.gc();  // se liberan dos objetos
      d2 = new Demo();
   }  // se liberan los objetos restantes
}
```

2.9. OBJETO QUE ENVIA EL MENSAJE: `this`

`this` es una referencia al objeto que envía un mensaje, o simplemente, una referencia al objeto que llama un método (este no debe ser `static`). Internamente se define:

```
final NombreClase this;
```

por consiguiente, no puede modificarse. Las variables y los métodos de las clases están referenciados, implícitamente, por `this`. Pensemos, por ejemplo, en la siguiente clase:

```
class Triangulo
{
   private double base;
   private double altura;
   public double area()
   {
      return base*altura/2.0 ;
   }
}
```

En el método `area()`se hace referencia a las variables instancia `base` y `altura`. ¿A la `base`, `altura` de qué objeto? El método es común para todos los objetos `Triangulo`. Aparentemente no distingue entre un objeto u otro, pero cada variable instancia implícitamente está cualificada por `this`. Es como si estuviera escrito:

```
public double area()
{
   return this.base*this.altura/2.0 ;
}
```

Fundamentalmente, `this` tiene dos usos:

- Seleccionar explícitamente un miembro de una clase con el fin de dar mas claridad o de evitar colisión de identificadores. Por ejemplo:

```
class Triangulo
{
   private double base;
   private double altura;
   public void datosTriangulo(double base, double altura)
   {
      this.base = base;
      this.altura = altura;
   }
   // ...
}
```

Se ha evitado con `this` la colisión entre argumentos y variables instancia.

- Que un método devuelva el mismo objeto que lo llamó. De esa manera, se pueden hacer llamadas en cascada a métodos de la misma clase. De nuevo se define la clase `Triangulo`:

```
class Triangulo
{
   private double base;
   private double altura;
   public Triangulo datosTriangulo(double base, double altura)
   {
      this.base = base;
      this.altura = altura;
      return this;
   }
   public Triangulo visualizar()
   {
      System.out.println(" Base = " + base);
      System.out.println(" Altura = " + altura);
      return this;
   }
   public double area()
   {
      return base*altura/2.0 ;
   }
}
```

Ahora se pueden concatenar llamadas a métodos:

```
Triangulo t = new Triangulo();
t.datosTriangulo(15.0,12.0).visualizar();
```

2.10. MIEMBROS `static` DE UNA CLASE

Cada instancia de una clase, cada objeto, tiene su propia copia de las variables de la clase. Cuando interese que haya miembros que no estén ligados a los objetos sino a la clase y, por tanto, sean comunes a todos los objetos, estos se declaran `static`.

2.10.1. Variables `static`

Las variables de clase `static` son compartidas por todos los objetos de la clase. Se declaran de igual manera que otra variable, añadiendo como prefijo la palabra reservada `static`. Por ejemplo:

```
public class Conjunto
{
   private static int k = 0;
   static Totem lista = null;
      // ...
}
```

Las variables miembro `static` no forman parte de los objetos de la clase sino de la propia clase. Dentro de las clase, se accede a ellas de la manera habitual, simplemente con su nombre. Desde fuera de la clase, se accede con el nombre de la clase, el selector y el nombre de la variable:

```
Conjunto.lista = ... ;
```

También se puede acceder a través de un objeto de la clase pero no es recomendable, ya que los miembros `static` no pertenecen a los objetos sino a las clases.

Ejercicio 2.3

Dada una clase, se quiere conocer en todo momento los objetos activos en la aplicación.

Se declara la clase `Ejemplo` con un constructor por defecto y otro con un argumento. Ambos incrementan la variable `static cuenta` en `1`. De esa manera, cada nuevo objeto queda contabilizado. También se declara el método `finalize()`, de tal forma que al activarse `cuenta` decrece en `1`.

El método `main()` crea objetos de la clase `Ejemplo` y visualiza la variable que contabiliza el número de sus objetos.

```
class Ejemplo
{
   private  int datos;
   static int cuenta = 0;

   public  Ejemplo()
   {
      datos = 0;
      cuenta++;     // nuevo objeto
   }
   public  Ejemplo(int g)
   {
      datos = g;
      cuenta++;     // nuevo objeto
   }
   // redefinición de finalize()
   protected void finalize()
   {
      System.out.println("Fin de objeto Ejemplo");
      cuenta--;
   }
}

public class ProbarEjemplo
{
   public static void main(String[] ar)
   {
      Ejemplo d1, d2;

      System.out.println("Objetos Ejemplo: " + Ejemplo.cuenta);
      d1 = new Ejemplo();
      d2 = new Ejemplo(11);
      System.out.println("Objetos Ejemplo: " + Ejemplo.cuenta);
```

```
      d2 = d1;
      System.gc();
      System.out.println("Objetos Ejemplo: " + Ejemplo.cuenta);

      d2 = d1 = null;
      System.gc();
      System.out.println("Objetos Ejemplo: " + Ejemplo.cuenta);
   }
}
```

Una variable `static` suele inicializarse directamente en la definición. Sin embargo, existe una construcción de Java que permite inicializar miembros `static` en un bloque de código dentro de la clase; el bloque debe venir precedido de la palabra `static`. Por ejemplo:

```
class DemoStatic
{
   private static int k;
   private static double r;
   private static String cmn;
   static
   {
      k = 1;
      r = 0.0;
      cmn = "Bloque";
   }
}
```

2.10.2. Métodos static

Los métodos de las clases se llaman a través de los objetos. En ocasiones interesa definir métodos que estén controlados por la clase, de modo que no haga falta crear un objeto para llamarlos: son los métodos `static`. Muchos métodos de la biblioteca Java están definidos como `static`; es, por ejemplo, el caso de los métodos matemáticos de la clase `Math`: `Math.sin(), Math.sqrt()`.

La llamada a los métodos `static` se realiza a través de la clase: `NombreClase.metodo()`, respetando las reglas de visibilidad. También se pueden llamar con un objeto de la clase; pero no es recomendable debido a que son métodos dependientes de la clase y no de los objetos.

Los métodos definidos como `static` no tienen asignada la referencia `this`, por lo que sólo pueden acceder a miembros `static` de la clase. Es un error que un método `static` acceda a miembros de la clase no `static`. Por ejemplo:

```
class Fiesta
{
   int precio;
   String cartel;
   public static void main(String[] a)
   {
      cartel = " Virgen de los pacientes";
      precio = 1;

      ...
```

al compilar da dos errores debido a que desde el método `main()`, definido como `static`, se accede a miembros no `static`.

Ejemplo 2.9

La clase `SumaSerie` *define tres variables* `static`*, y un método* `static` *que calcula la suma cada vez que se llama.*

```
class SumaSerie
{
   private static long n;
   private static long m;
   static
   {
      n = 0;
      m = 1;
   }
   public static long suma()
   {
      m += n;
      n = m - n;
      return m;
   }
}
```

2.11. CLASE `Object`

`Object` es la superclase base de todas las clases de Java; toda clase definida en Java hereda de la clase `Object` y, en consecuencia, toda variable referencia a una clase se convierte, automáticamente, al tipo `Object`. Por ejemplo:

```
Object g;
String cd = new String("Barranco la Parra");
Integer y = new Integer(72); // objeto   inicializado a 72
g = cd; // g referencia al mismo objeto que cd
g = y; // g ahora referencia a un objeto Integer
```

La clase `Object` tiene dos métodos importantes: `equals()` y `toString()`. Generalmente, se redefinen en las clases para especializarlos.

equals()

Compara el objeto que hace la llamada con el objeto que se pasa como argumento, devuelve `true` si son iguales.

```
boolean equals(Object k);
```

El siguiente ejemplo compara dos objetos; la comparación es `true` si contienen la misma cadena.

```
String ar = new String("Iglesia románica");
```

```
String a = "Vida sana";
if (ar.equals(a)) //...no se cumple, devuelve false
```

toString()

Este método construye una cadena, que es la representación del objeto, y devuelve la cadena. Normalmente, se redefine en las clases para dar así detalles explícitos de los objetos de la clase.

```
String toString()
```

Por ejemplo, un objeto `Double` llama al método `toString()` y asigna la cadena a una variable.

```
Double r = new Double(2.5);
String rp;
rp = r.toString();
```

2.11.1. Operador `instanceof`

Con frecuencia, se necesita conocer la clase de la que es instancia un objeto. Hay que tener en cuenta que, en las jerarquías de clases, se dan conversiones automáticas entre clases derivadas y su clase base; en particular, cualquier referencia se puede convertir a una variable de tipo `Object`.

Con el operador `instanceof` se determina la clase a la que pertenece un objeto, que tiene dos operandos: el primero es un objeto y, el segundo, una clase. Evalúa la expresión a `true` si el primero es una instancia del segundo. La siguiente función tiene una argumento de tipo `Object`, por lo que puede recibir cualquier referencia, seleccionando la clase a la que pertenece el objeto transmitido (`String`, `Vector`,...):

```
public hacer (Object g)
{
   if (g instanceof String)
    ...
   else if (g instanceof Vector)
    ...
```

> El operador `instanceof` es un operador relacional, su evaluación da como resultado un valor de tipo `boolean`.

2.12. TIPOS ABSTRACTOS DE DATOS EN JAVA

La implementación de un TAD en Java se realiza de forma natural con una clase. Dentro de la clase va a residir la representación de los datos junto a las operaciones (métodos de la clase). La interfaz del tipo abstracto queda perfectamente determinada con la etiqueta `public`, que se aplicará a los métodos de la clase que representen operaciones.

Por ejemplo, si se ha especificado el TAD `Punto` para representar la abstracción punto en el espacio tridimensional, la siguiente clase implementa el tipo:

```
class Punto
{
```

```
    // representación de los datos
    private double x, y, z;
    // operaciones
    public double distancia(Punto p);
    public double modulo();
    public double anguloZeta();
       ...
};
```

2.12.1. Implementación del TAD Conjunto

La implementación de un TAD se realiza según la especificación realizada del tipo. La clase `Conjunto` implementa el *TAD* `Conjunto`, cuya especificación se encuentra en el apartado 2.3. La clase representa los datos de forma genérica, utiliza un array para almacenar los elementos, de tipo `Object`.

Archivo conjunto.java

```
package conjunto;

public class Conjunto
{
   static int M = 20;    // aumento de la capacidad
   private Object [] cto;
   private int cardinal;
   private int capacidad;
   // operaciones
   public Conjunto()
   {
      cto = new Object[M];
      cardinal = 0;
      capacidad = M;
   }
   // determina si el conjunto está vacío
   public boolean esVacio()
   {
      return (cardinal == 0);
   }
   // añade un elemento si no está en el conjunto
   public void annadir(Object elemento)
   {
      if (!pertenece(elemento))
      {
         /* verifica si hay posiciones libres,
            en caso contrario amplia el conjunto */
         if (cardinal == capacidad)
         {
            Object [] nuevoCto;
            nuevoCto = new Object[capacidad + M];
            for (int k = 0; k < capacidad; k++)
              nuevoCto[k] = cto[k];
            capacidad += M;
            cto = nuevoCto;
            System.gc();       // devuelve la memoria no referenciada
```

```
        }
        cto[cardinal++] = elemento;
      }
   }
   // quita elemento del conjunto
   public void retirar(Object elemento)
   {
      if (pertenece(elemento))
      {
         int k = 0;
         while (!cto[k].equals(elemento))
           k++;
         /* desde el elemento k hasta la última posición
            mueve los elementos una posición a la izquierda */
         for (; k < cardinal ; k++)
           cto[k] = cto[k+1];
         cardinal--;
      }
   }
   //busca si un elemento pertenece al conjunto
   public boolean pertenece(Object elemento)
   {
      int k = 0;
      boolean encontrado = false;
      while (k < cardinal && !encontrado)
      {
         encontrado = cto[k].equals(elemento);
         k++;
      }
      return encontrado;
   }
   //devuelve el número de elementos
   public int cardinal()
   {
      return this.cardinal;
   }
   //operación unión de dos conjuntos
   public Conjunto union(Conjunto c2)
   {
      Conjunto u = new Conjunto();
         // primero copia el primer operando de la unión
      for (int k = 0; k < cardinal; k++)
         u.cto[k] = cto[k];
      u.cardinal = cardinal;
         // añade los elementos de c2 no incluidos
      for (int k = 0; k < c2.cardinal; k++)
         u.annadir(c2.cto[k]);
      return u;
   }
   public Object elemento(int n) throws Exception
   {
      if (n <= cardinal)
         return cto[--n];
      else
         throw new Exception("Fuera de rango");
   }
}
```

RESUMEN

Los tipos abstractos de datos (TAD) describen un conjunto de objetos con la misma representación y comportamiento. Los tipos abstractos de datos presentan una separación clara entre la interfaz externa de un tipo de datos y su implementación interna. La implementación de un tipo abstracto de datos está oculta. Por consiguiente, se pueden utilizar implementaciones alternativas para el mismo tipo abstracto de dato sin cambiar su interfaz.

La especificación de un tipo abstracto de datos se puede hacer de manera *informal,* o bien, de forma mas rigurosa, una especificación *formal*. En la especificación *informal* se describen literalmente los datos y la funcionalidad de las operaciones. La especificación formal describe los datos, la sintaxis y la semántica de las operaciones, considerando ciertas operaciones como axiomas, que son los constructores de nuevos datos. Una buena especificación formal de un tipo abstracto de datos debe poder verificar la *bondad* de la implementación.

En la mayoría de los lenguajes de programación orientados a objetos, y en particular en Java, los tipos abstractos de datos se implementan mediante **clases.**

Una **clase** es un tipo de dato definido por el programador que sirve para representar objetos del mundo real. Un objeto de una clase tiene dos componentes: un conjunto de atributos o variables instancia y un conjunto de comportamientos (métodos). Los atributos también se llaman variables instancia o miembros dato, y los comportamientos se llaman métodos miembro.

```
class Circulo
{
   private double centroX;
   private double centroY;
   private double radio;
   public double superficie(){}
}
```

Un objeto es una instancia de una clase, y una variable cuyo tipo sea la clase es una referencia a un objeto de la clase.

```
Circulo unCirculo; // variable del tipo clase
Circulo [] tipocirculo = new Circulo[10]; // array de referencias
```

La definición de una clase, en cuanto a visibilidad de sus miembros, tiene tres secciones: *pública, privada y protegida*. La sección pública contiene declaraciones de los atributos y del comportamiento del objeto que son accesibles a los usuarios del objeto. Se recomienda la declaración de los constructores en la sección pública. La sección privada contiene los métodos miembro y los miembros dato, que son ocultos o inaccesibles a los usuarios del objeto. Estos métodos miembro y atributos dato son accesibles sólo por los métodos miembro del objeto. Los miembros de una clase con visibilidad `protected` son accesibles para cualquier usuario de la clase que se encuentre en el mismo `package`; también son accesibles para las clases derivadas. El acceso *por defecto,* sin modificador, tiene las mismas propiedades que el acceso `protected` para las clases que se encuentran en el mismo `package`.

Un **constructor** es un método miembro con el mismo nombre que su clase. Un constructor no puede devolver un tipo pero puede ser sobrecargado.

```
class complejo
{
   public complejo(double x, double y){}
   public complejo(complejo z){}
}
```

El **constructor** es un método especial que se invoca cuando se crea un objeto. Se utiliza, normalmente, para inicializar los atributos de un objeto. Por lo general, al menos se define un constructor sin argumentos, llamado constructor por defecto. En caso de no definirse el constructor, implícitamente queda definido un constructor sin argumentos que inicializa cada miembro numérico a `0`, los miembros de tipo `boolean` a `true` y las referencias a `null`.
El proceso de crear un objeto se llama *instanciación* (creación de instancia). En Java se crea un objeto con el operador `new` y un constructor de la clase.

```
Circulo C = new Circulo();
```

En Java, la liberación de objetos es automática; cuando un objeto deja de estar referenciado por una variable es candidato a que la memoria que ocupa sea liberada y, posteriormente, reutilizada. El proceso se denomina *garbage collection*, el método `System.gc()` realiza el proceso.

Los paquetes son agrupaciones de clases relativas a un tema. El sistema suministra paquetes con clases que facilitan la programación. Se puede afirmar que el paquete *java.lang* es donde se encuentran las clases mas utilizadas, por lo que es automáticamente incorporado a los programas.

Los miembros de una clase definidos como `static` no están ligados a los objetos de la clase sino que son comunes a todos los objetos, son de la clase. Se cualifican con el nombre de la clase, por ejemplo:

```
Math.sqrt(x);
```

EJERCICIOS

2.1. Realizar una especificación informal del TAD `Conjunto` con las operaciones: *ConjuntoVacio, Esvacio, Añadir* un elemento al conjunto, *Pertenece* un elemento al conjunto, *Retirar* un elemento del conjunto, *Union* de dos conjuntos, *Intersección* de dos conjuntos e *Inclusión* de conjuntos.

2.2. Realizar la especificación formal del TAD `Conjunto` con las operaciones indicadas en el Ejercicio 2.1.

Considerar las operaciones *ConjuntoVacio* y *Añadir* como constructores.

2.3. Construir el TAD Natural para representar los números naturales, con las operaciones: *Cero, Sucesor, EsCero, Igual, Suma, Antecesor, Diferencia* y *Menor.*

Realizar la especificación informal y formal considerando como constructores las operaciones *Cero* y *Sucesor.*

2.4. Diseñar el TAD Bolsa como una colección de elementos no ordenados y que pueden estar repetidos. Las operaciones del tipo abstracto son *CrearBolsa*, *Añadir* un elemento, *BolsaVacia* (verifica si tiene elemento), *Dentro* (verifica si un elementos pertenece a la bolsa), *Cuantos* (determina el número de veces que se encuentra un elemento), *Union* y *Total*.

Realizar la especificación informal y formal considerando como constructores las operaciones *CrearBolsa* y *Añadir.*

2.5. Diseñar el TAD Complejo para representar los números complejos. Las operaciones que se deben definir son *AsignaReal* (asigna un valor a la parte real), *AsignaImaginaria* (asigna un valor a la parte imaginaria), *ParteReal* (devuelve la parte real de un complejo), *ParteImaginaria* (devuelve la parte imaginaria de un complejo), *Modulo* de un complejo y *Suma* de dos números complejos. Realizar la especificación informal y formal considerando como constructores las operaciones que desee.

2.6. Diseñar el tipo abstracto de datos Matriz con la finalidad de representar matrices matemáticas. Las operaciones a definir son *CrearMatriz* (crea una matriz, sin elementos, de `m` filas por `n` columnas), *Asignar* (asigna un elemento en la fila `i`, columna `j`), *ObtenerElemento* (obtiene el elemento de la fila `i`, y columna `j`), *Sumar* (realiza la suma de dos matrices cuando tienen las mismas dimensiones), *ProductoEscalar* (obtiene la matriz resultante de multiplicar cada elemento de la matriz por un valor). Realizar la especificación informal y formal considerando como constructores las operaciones que desee.

2.7. ¿Qué está mal en la siguiente definición de la clase ?

```
import java.io.*;

class Buffer
{
   private char datos[];
   private int cursor ;
   private Buffer(int n)
   {
      datos = new char[n]
   };
   public static int Long( return cursor;);
   public String contenido(){}
}
```

2.8. Dado el siguiente programa, ¿es legal la sentencia de `main()`?

```
class Punto
{
   public int x, int y;
   public Punto(int x1, int y1) {x = x1 ; y = y1;}
}

class CreaPunto
{
   public static void main(String [] a)
   {
```

```
    new Punto(25, 15); //¿es legal esta sentencia?
    Punto p = new Punto(); //¿es legal esta sentencia?
    }
}
```

2.9. Suponiendo contestado el ejercicio anterior, ¿cuál será la salida del siguiente programa?

```
class CreaPunto
{
   public static void main(String [] a)
   {
     Punto q;
     q = new Punto(2, 1);
     System.out.println("x = " + p.x + "\ty = " + p.y);
```

2.10. Dada la siguiente clase, escribir el método `finalize()` y un programa que cree objetos, después pierda las referencias a los objetos creados y se active el método `finalize()`.

```
class Operador
{
   public float memoria;
   public Operador(void)
   {
      System.out.println("Activar maquina operador");
      memoria = 0.0F;
   }

   public float sumar(float f)
   {
      memoria += f;
      return memoria;
   }
}
```

2.11. Se desea realizar una clase `Vector3d` que permita manipular vectores de tres componentes (coordenadas `x, y, z`) de acuerdo con las siguientes normas:

- Sólo posee un método constructor.
- Tiene un método miembro `equals()`que permite saber si dos vectores tienen sus componentes o coordenadas iguales.

2.12. Incluir en la clase `Vector3d` del Ejercicio 2.11 el método `normamax` que permita obtener la norma de dos vectores (Nota: La norma de un vector $v = x, y, z$ es $\sqrt{x^2 + y^2 + z^2}$).

2.13. Realizar la clase `Complejo` que permita la gestión de números complejos (un número complejo = dos números reales `double`: una parte real + una parte imaginaria). Las operaciones a implementar son las siguientes:

- `establecer()` permite inicializar un objeto de tipo `Complejo` a partir de dos componentes `double`.

- `imprimir()` realiza la visualización formateada de un `Complejo`.
- `agregar()` (sobrecargado) para añadir, respectivamente, un `Complejo` a otro y añadir dos componentes `double` a un `Complejo`.

2.14. Añadir a la clase `Complejo` del Ejercicio 2.13 las siguientes operaciones:

- Suma: `a + c = (A+C,(B+D)i).`
- Resta: `a - c = (A-C,(B-D)i).`
- Multiplicación: `a*c = (A*C-B*D,(A*D+B*C)i)`
- Multiplicación: `x*c = (x*C, x*Di), donde x es real.`
- Conjugado: `~a = (A ,-Bi).`

  Siendo `a = A+Bi; c = C+Di`

2.15. Implementar la clase `Hora`. Cada objeto de esta clase representa una hora específica del día, almacenando las horas, minutos y segundos como enteros. Se ha de incluir un constructor, métodos de acceso, una método `adelantar(int h, int m, int s)` para adelantar la hora actual de un objeto existente, un método `reiniciar(int h, int m, int s)` que reinicializa la hora actual de un objeto existente y un método `imprimir()`.

PROBLEMAS

2.1. Implementar el TAD `Bolsa` descrito en el Ejercicio 2.4. Probar la implementación con un programa que invoque a las operaciones del tipo abstracto Bolsa.

2.2. Implementar el TAD `Matriz` especificado en el Ejercicio 2.6. Escribir un programa que haciendo uso del tipo `Matriz` realice operaciones diversas (lectura, suma...) y escriba las matrices generadas.

2.3. Implementar la clase `Fecha` con miembros dato para el mes, día y año. Cada objeto de esta clase representa una fecha, que almacena el día, mes y año como enteros. Se debe incluir un constructor por defecto, métodos de acceso, un método `reiniciar (int d, int m, int a)` para reiniciar la fecha de un objeto existente, un método `adelantar(int d, int m, int a)` para avanzar a una fecha existente (dia, `d`; mes, `m`; y año `a`) y un método `imprimir()`. Escribir un método de utilidad, `normalizar()`, que asegure que los miembros dato están en el rango correcto `1 ≤ año, 1 ≤ mes ≤ 12, día ≤ días (mes)`, donde `días(Mes)` es otro método que devuelve el número de días de cada mes.

2.4. Ampliar el programa anterior de modo que pueda aceptar años bisiestos.

Nota: un año es bisiesto si es divisible por 400, o si es divisible por 4 pero no por 100. Por ejemplo, los años 1992 y 2000 son años bisiestos y 1997 y 1900 no son bisiestos.



CAPITULO 3

Arrays (arreglos) y cadenas

Objetivos

Una vez que se haya leído y estudiado este capítulo, usted podrá:

- Organizar colecciones de ítems en una misma estructura de programación.
- Diferenciar entre un tipo simple y un tipo estructurado.
- Declarar variables array de una dimensión y distinguir una declaración de definición.
- Aplicar el operador `new` para determinar el máximo número de elementos de un array.
- Escribir cadenas constantes en los programas Java.
- Declarar variables `String` e inicializarlas a una cadena constante.
- Conocer las distintas operaciones de la clase `String`.
- Definir no sólo arrays de una dimensión, sino de dos o mas dimensiones.
- Pasar arrays a un método y distinguir paso por valor y paso por referencia.

Contenido

Conceptos clave

- ♦ Array.
- ♦ Arrays bidimensionales.
- ♦ Arrays multidimensionales.
- ♦ Cadena de texto.
- ♦ Declaración de un array.
- ♦ Lista, tabla.

Para profundizar (página web: www.mhe.es/joyanes)

- Cadenas de tipo `StringBuffer`.

INTRODUCCIÓN

En este capítulo se examinará el tipo *array* (lista o tabla). Aprenderá el concepto y tratamiento de los *arrays*. Un *array* almacena muchos elementos del mismo tipo, tales como veinte enteros, cincuenta números de coma flotante o quince caracteres. Los *arrays* pueden ser de una dimensión vectores, que son los mas utilizados ; de dos dimensiones tablas o matrices ; también de tres o más dimensiones. Java proporciona la clase `String`, que representa una secuencia de caracteres y las operaciones con cadenas más comunes, y también dispone de la clase especializada `StringBuffer` para procesar cadenas que pueden sufrir cambios. Una *cadena* se considera como un objeto de la clase `String` que no puede modificarse.

3.1. *ARRAYS* (ARREGLOS)

Un *array* o *arreglo*[1] (lista o tabla) es una secuencia de datos del mismo tipo. Los datos se llaman elementos del *array* y se numeran consecutivamente `0, 1, 2, 3 ...` El tipo de elementos almacenados en el *array* puede ser cualquier dato simple de Java o de un tipo previamente declarado como una clase. Normalmente, el *array* se utiliza para almacenar tipos tales como `char`, `int` o `float`.

Un *array* puede contener, por ejemplo, la edad de los alumnos de una clase, las temperaturas de cada día de un mes en una ciudad determinada o el número de personas que residen en cada una de las diecisiete comunidades autónomas españolas. Cada ítem del *array* se denomina *elemento*.

Los elementos de un *array* se numeran, como ya se ha comentado, consecutivamente `0, 1, 2, 3,...` Estos números se denominan *valores índice* o *subíndice* del *array*. El término "subíndice" se utiliza ya que especifica, igual que en matemáticas, una secuencia tal como a_0, a_1, a_2... Estos números localizan la posición del elemento dentro del *array*, proporcionando *acceso directo* al *array*.

Si el nombre del *array* es `a`, entonces `a[0]` es el nombre del elemento que está en la posición `0`, `a[1]` es el nombre del elemento que está en la posición `1`, etc. En general, el elemento *i-ésimo* está en la posición *i-1*, de modo que si el *array* tiene *n* elementos, sus nombres son `a[0], a[1],...,a[n-1]`. Gráficamente, se representa así el *array* `a` con seis elementos.

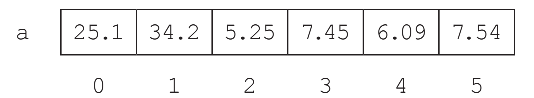


Figura 3.1 *Array* de seis elementos

El *array* `a` tiene 6 elementos: `a[0]` contiene `25.1`, `a[1]` contiene `34.2`, `a[2]` contiene `5.25`, `a[3]` contiene `7.45`, `a[4]` contiene `6.09` y `a[5]` contiene `7.54`. El diagrama de la Figura 3.1 representa realmente una región de la memoria de la computadora, ya que un *array* se almacena siempre con sus elementos en una secuencia de posiciones de memoria contigua.

En Java, los índices de un *array* siempre tienen como límite inferior `0` y como índice superior el tamaño del *array* menos `1`.

3.1.1. Declaración de un *array*

Un *array* se declara de modo similar a otros tipos de datos, excepto que se debe indicar al compilador que es un *array,* lo que se hace con los corchetes.

```
int [] v;
float w[];
```

[1] En latinoamérica, el término inglés *array* se suele traducir por el término español *arrreglo*. En España, se suele utilizar el término en su acepción en inglés.

Los corchetes se pueden colocar de dos formas:

- A continuación del tipo de datos.
- A continuación del nombre del *array*.

Así, la sintaxis de declaración de variables *array* en Java es:

```
tipo [] identificador;
tipo identificador[];
```

El primer formato indica que todos los identificadores son *arrays* del tipo. En el segundo formato, *array* es sólo el identificador al que le siguen los `[]`.

Ejemplo 3.1

Se escriben distintas declaraciones de arrays.

1. `char cad[], p;`

   `cad` es un *array* de tipo `char`. `p` es una variable de tipo `char`.
2. `int [] v, w;`

   tanto `v` como `w` son declarados *arrays* unidimensionales de tipo `int`.
3. `double [] m, t[], x;`

   `m` y `x` son *array* de tipo `double`; `t` es un *array* de *array* con elementos de tipo `double`.

Precaución

Java no permite indicar el número de elementos en la declaración de una variable array. Por ejemplo, la declaración `int numeros[12]` el compilador producirá un error.

3.1.2. Creación de un *array*

Java considera que un *array* es una referencia a un objeto. En consecuencia, para que realmente se cree *(instancie)* el *array*, usa el operador `new` junto al tipo de los elementos del *array* y su número. Por ejemplo, para crear un *array* que guarde las notas de la asignatura de música en un aula de 26 alumnos:

```
float [] notas;
notas = new float[26];
```

Se puede escribir en una misma sentencia:

```
float [] notas = new float[26];
```

La *sintaxis* para declarar y definir un *array* de un número de elementos determinado es:

```
tipo nombreArray[] = new tipo[numeroDeElementos];
```

o bien,

```
tipo nombreArray[];
nombreArray = new tipo[numeroDeElementos];
```

Ejemplo 3.2

Se declaran y se crean arrays de diferentes tipos de datos.

1. `int a[] = new int [10];`
   `a` es un *array* de `10` elementos de tipo `int`.
2. ```
   final int N = 20;
   float [] vector;
   vector = new float[N];
   ```

Se ha creado un *array* de `N` elementos de tipo `float`. Para acceder al tercer elemento y leer un valor de entrada:

```
vector[2] = (Float.valueOf(entrada.readLine())).floatValue();
```

> ***Precaución***
>
> Es un error frecuente acceder a un elemento de un *array* fuera del rango en que está definido. Java comprueba en tiempo de compilación que los índices estén dentro de rango, en caso contrario genera un error. Durante la ejecución del programa, un acceso fuera de rango genera una excepción .

3.1.3. Subíndices de un *array*

El índice de un *array* se denomina, con frecuencia, *subíndice del* array. El término procede de las matemáticas, en las que un subíndice se utiliza para representar un elemento determinado.

$numeros_0$	*equivale a*	`numeros[0]`
$numeros_3$	*equivale a*	`numeros[3]`

El método de numeración del elemento *i-ésimo* con el índice o subíndice *i-1* se denomina *indexación basada en cero*. Se utiliza para que el índice de un elemento del *array* sea siempre igual que el número de "*pasos*" desde el elemento inicial `numeros[0]` a ese elemento. Por ejemplo, `numeros[3]` está a 3 pasos o posiciones del elemento `numeros[0]`.

Ejemplo 3.3

Acceso a elementos de diferentes arrays.

1. `int []mes = new int[12];` — `mes` *contiene 12 elementos: el primero,* `mes[0]`, *y el último,* `mes[11]`.

   `float salarios[];` — *Declara un array de tipo* `float`.
   `salarios = new float[25];` — *Crea el array de 25 elementos.*

   `mes[4] = 5;`
   `salario[mes[4]*3];` — *Accede al elemento* `salario[15]`.

2. `final int MX = 20;`
   `Racional []ra = new Racional[MX];` — *Declara un array de 20 objetos* `Racional`.
   `ra[MX - 4];` — *Accede al elemento* `ra[16]`.
```

En los programas se pueden referenciar elementos del *array* utilizando fórmulas para los subíndices. Siempre que el subíndice pueda evaluarse a un entero, se puede utilizar una constante, una variable o una expresión para el subíndice.

3.1.4. Tamaño de los *arrays*. Atributo `length`

Java considera cada *array* como un objeto que, además de tener capacidad para almacenar elementos, dispone del atributo `length` con el número de elementos.

```
double [] v = new double[15];
System.out.print(v.length); //escribe 15, número de elementos de v.
```

Java conoce el número de elementos de un *array* cuando se establece su tamaño con el operador `new`, o bien con una expresión de inicialización. `length` está protegido, no puede ser modificado ya que está definido con el cualificador `final`.

Ejemplo 3.4

Haciendo uso del atributo `length` *se calcula la suma de los elementos de un array de tipo* `double`.

```
double suma (double [] w)
{
   double s = 0.0;
   for (int i = 0; i < w.length; i++)
      s += w[i];
   return s;
}
```

Precaución

El número de elementos de un array es un campo del array, no un método:

```
w.length;   // es correcto
w.length(); // es un error
```

3.1.5. Verificación del índice de un *array*

Java, al contrario que el lenguaje C, verifica que el índice de un *array* esté en el rango de definición. Si, por ejemplo, se define un *array* `a` de 6 elementos, los índices válidos están en el rango 0 a 5, entonces el acceso `a[6]` es detectado por el compilador y genera un mensaje de error. Durante la ejecución del programa también puede producirse el acceso a un elemento fuera de los índices, y provocará que el programa se *"rompa"* en tiempo de ejecución, generando una excepción.

Ejemplo 3.5

Protección frente a errores en el intervalo (rango) de valores de una variable de índice que representa un array

```
int datos(double a[])throws Exception
{
```

```
      int n;
      System.out.println("Entrada de datos, cuantos elementos: ?  ");
      n = Integer.parseInt(entrada.readLine());
      if (n > a.length)
         return 0;
      for (int i = 0; i < n; i++)
         a[i]= Double.valueOf(entrada.readLine()).doubleValue();
       return 1;
   }
```

3.1.6. Inicialización de un *array*

Los elementos del *array* se pueden inicializar con valores constantes en una sentencia que, además, determina su tamaño. Estas constantes se separan por comas y se encierran entre llaves, como en los siguientes ejemplos:

```
int numeros[] = {10, 20, 30, 40, 50, 60}; /* Define un array de 6 elementos
                y se inicializan a las constantes */
int n[] = {3, 4, 5}          // Define un array de 3 elementos
char c[] = {'L','u','i','s'};  // Define un array de 4 elementos
```

El *array* `numeros` tiene `6` elementos, `n` tiene `3` elementos y el *array* `c` `4` elementos.

Nota

La serie de valores entre llaves sólo puede ser usada para inicializar un array, no en sentencias de asignación posteriores.

```
int cuenta[] = {15, 25, -45, 0, 50};
```

El compilador asigna automáticamente cinco elementos a `cuenta`.

El método de inicializar *arrays* mediante valores constantes después de su definición es adecuado cuando el número de elementos del *array* es pequeño. Por ejemplo:

```
final int ENE = 31, FEB = 28, MAR = 31, ABR = 30, MAY = 31,
          JUN = 30, JUL = 31, AGO = 31, SEP = 30, OCT = 31,
          NOV = 30, DIC = 31;

int meses[] = {ENE, FEB, MAR, ABR, MAY, JUN,
                              JUL, AGO, SEP, OCT, NOV, DIC};
```

Pueden asignarse valores a un *array* utilizando un bucle `for` o `while/do-while`, y éste suele ser el sistema más empleado normalmente. Por ejemplo, para inicializar todos los valores del *array* `numeros` al valor `-1` se puede utilizar la siguiente sentencia:

```
for (i = 0; i < numeros.length; i++)
   numeros[i] = -1;
```

Por defecto, Java inicializa cada elemento de un *array* a ceros binarios, ya sea de tipo `int`, `char, ...`

Ejercicio 3.1

El programa escrito a continuación lee NUM *enteros en un array, multiplica los elementos del array y visualiza el producto.*

```
import java.io.*;

class Inicial
{

   public static void main(String [] a) throws IOException
   {
      final int NUM = 10;
      BufferedReader entrada = new BufferedReader(
                              new InputStreamReader(System.in));
      int nums[]= new int[NUM];
      int total = 1;
      System.out.println("Por favor, introduzca " + NUM + " datos");
      for (int i = 0; i < NUM; i++)
      {
         nums[i] = Integer.parseInt(entrada.readLine());
      }
      System.out.print("\nLista de números: ");
      for (int i = 0; i < NUM; i++)
      {
         System.out.print(" " + nums[i]);
         total *= nums[i];
      }
      System.out.println("\nEl producto de los números es " + total);
   }
}
```

3.1.7. Copia de *arrays*

En Java, los nombres de *arrays* son referencias a un bloque de memoria distribuida según el número de elementos; por ello, si se hace una asignación entre dos variables *array,* éstas se refieren al mismo *array*. Por ejemplo:

```
double [] r, w;
r = new double[11];
w = new double[15];
for (int j = 0; j < r.length; j++)
   r[j] = (double) 2*j-1;
// asignación del array r a w
w = r;
```

Esta asignación hace que se pueda acceder a los elementos desde `r` o desde `w`, pero no se ha creado un nuevo almacenamiento para los elementos; los `15` elementos que inicialmente se referencian desde `w` se han perdido.

Los elementos de un *array* se pueden asignar a otro *array* del mismo tipo. Se construye un bucle que accede a cada elemento del origen y destino; el *array* destino debe estar definido con al menos los mismo elementos. Por ejemplo:

```
final int N = 12;
int v1[] = new int[N], v2[] = new int[N];

for (int i = 0; i < N; i++)
   v1[i] = (int)Math.random()*199 +1 ;
// Los elementos de v1 son copiados a v2
for (int i = 0; i < N; i++)
   v2[i] = v1[i];
```

Esta copia se puede hacer con un método de la clase `System`, `arraycopy()`. Para copiar los `N` elementos que tiene el *array* `v1` en `v2` con el método `arraycopy()` se especifica la posición inicial del vector desde el que se copia, la posición del vector destino donde se inicia la copia y el número de elementos:

```
System.arraycopy(v1,0,v2,0,N);
```

La sintaxis del método `arraycopy`:

```
System.arraycopy(arrayOrigen,inicioOrigen,arrayDestino,inicioDestino,numElementos)
```

`arrayOrigen:`	nombre del array desde el que se va a copiar.
`inicioOrigen:`	posición del array origen desde el que se inicia la copia.
`arrayDestino:`	nombre del array en el que se hace la copia.
`inicioDestino:`	posición del array destino donde empieza la copia.
`numElementos:`	número de elementos del array origen que se van a copiar.

Ejercicio 3.2

Definir dos arrays de tipo double, v y w con 15 y 20 elementos respectivamente. En el array v se guardan los valores de la función e^{2x-1} para $x \geq 1.0$; el array w se inicializa cada elemento al ordinal del elemento. A continuación se copian los 10 últimos elementos de v a partir del elemento 11 de w. Por último, se escriben los elementos de ambos arrays.

El programa que se escribe a continuación sigue los pasos indicados en el enunciado. Se usa la función `exp()` de la clase `Math` para el cálculo de la función e^{2x-1}; así como el método `arraycopy()` para realizar la copia de elementos de *array* pedida.

```
import java.io.*;
class copiArray
{
   public static void main(String [] a)
   {
      final int N = 15;
      final int M = 20;
      double [] v = new double[N], w = new double [M];
      double x = 1.0;

      for (int i = 0; i < N; x += 0.2, i++)
         v[i] =  Math.exp(2*x-1);
```

```
        for (int i = 0; i < M; i++)
           w[i] = (double)i;
           // Se imprimen los elementos del vector v
        System.out.println("\n Valores del vector v");
        for (int i = 0; i < N; i++)
           System.out.print(v[i] + " ");
        System.out.flush();
           // Es realizada la copia de v a w
        System.arraycopy(v, (N-1)-10 +1, w, 10, 10);
        // Se impimen los elementos del vector w
        System.out.println("\n Valores del vector w");
        for (int i = 0; i < M; i++)
           System.out.print(w[i] + " ");
        System.out.flush();
     }
  }
```

Precaución

Ha de haber espacio suficiente en el array destino para realizar la copia de elementos desde el array fuente; en caso contrario, se provoca un error en la ejecución.

3.2. *ARRAYS* MULTIDIMENSIONALES

Los *arrays* vistos anteriormente se conocen como *arrays unidimensionales* (una sola dimensión) y se caracterizan por tener un solo subíndice. Estos *arrays* se conocen también por el término *listas*. Los *arrays multidimensionales* son aquellos que tienen más de una dimensión y, en consecuencia, más de un índice. Los más usuales son los de dos dimensiones, conocidos también por el nombre de *tablas* o *matrices*. Sin embargo, es posible crear *arrays* de tantas dimensiones como requieran sus aplicaciones, ya sean tres, cuatro o más.

Un *array* de dos dimensiones *(m × n)* equivale a una tabla con múltiples filas y múltiples columnas (Figura 3.2).

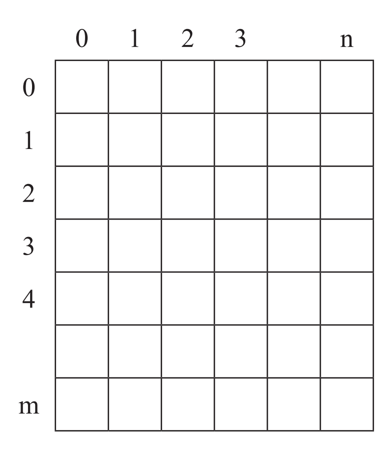


Figura 3.2 Estructura de un *array* de dos dimensiones

En el *array* bidimensional de la Figura 3.2, si las filas se etiquetan de `0` a `m` y las columnas de `0` a `n`, el número de elementos que tendrá el *array* será el resultado del producto `(m+1)*(n+1)`. El sistema de localizar un elemento es por las coordenadas representadas por su número de fila y su número de columna *(a, b)*. La sintaxis para la declaración de un *array* de dos dimensiones es:

```
<tipo de datoElemento> <nombre array> [][];
```

o bien

```
<tipo de datoElemento> [][]<nombre array>;
```

Ejemplos de declaración de matrices :

```
char pantalla[][];
int puestos[][];
double [][]matriz;
```

Estas declaraciones no reservan memoria para los elementos de la matriz, realmente son referencias. Para reservar memoria y especificar el número de filas y de columnas se utiliza el operador `new`. Así, a partir de las declaraciones anteriores:

```
pantalla  = new char[80][24]; // matriz con 80 filas y 24 columnas
puestos = new int[10][5];      // matriz de 10 filas por 5 columnas
final int N = 4;
matriz = new double[N][N];    // matriz cuadrada de N*N elementos
```

El operador `new` se puede aplicar a la vez que se hace la declaración. La sintaxis para definir una matriz es:

```
<tipo de datoElemento> <nombre array>[][]=
      new <tipo de datoElemento> [<NúmeroDeFilas<] [<NúmeroDeColumnas>];
```

Atención

Java requiere que cada dimensión esté encerrada entre corchetes. La sentencia `int equipos[][] = new int[4,5]` no es válida.

Un *array* de dos dimensiones es en realidad un *array de arrays*, es decir, un *array* unidimensional, y cada elemento no es un valor entero, de coma flotante o carácter, sino que cada elemento es otro *array.*

Los elementos de los *arrays* se almacenan en la memoria de modo que el subíndice más próximo al nombre del *array* es la fila y el otro subíndice, la columna. La Tabla 3.1 presenta todos los elementos y sus posiciones relativas en la memoria del *array,* `int [][]tabla = new int[4][2]`.

Tabla 3.1 Un *array* bidimensional

Elemento	Posición relativa de memoria
`tabla[0][0]`	0
`tabla[0][1]`	4

(*Continúa*)

Elemento	Posición relativa de memoria
`tabla[1][0]`	8
`tabla[1][1]`	12
`tabla[2][0]`	16
`tabla[2][1]`	20
`tabla[3][0]`	24
`tabla[3][1]`	28

3.2.1. Inicialización de *arrays* multidimensionales

La inicialización se hace encerrando entre llaves la lista de constantes, separadas por comas, que forma cada fila, como en los ejemplos siguientes:

1. `int tabla1[][] = { {51, 52, 53},{54, 55, 56} };`

   Define una matriz de 2 filas por 3 columnas cada una.
   O bien con este formato más *amigable*:

```
int tabla1[][] = {   {51, 52, 53},
                     {54, 55, 56} };
```

2.
```
int tabla2[][] = {
                    {1, 2, 3, 4},
                    {5, 6, 7, 8},
                    {9, 10, 11, 12}
                 };
```

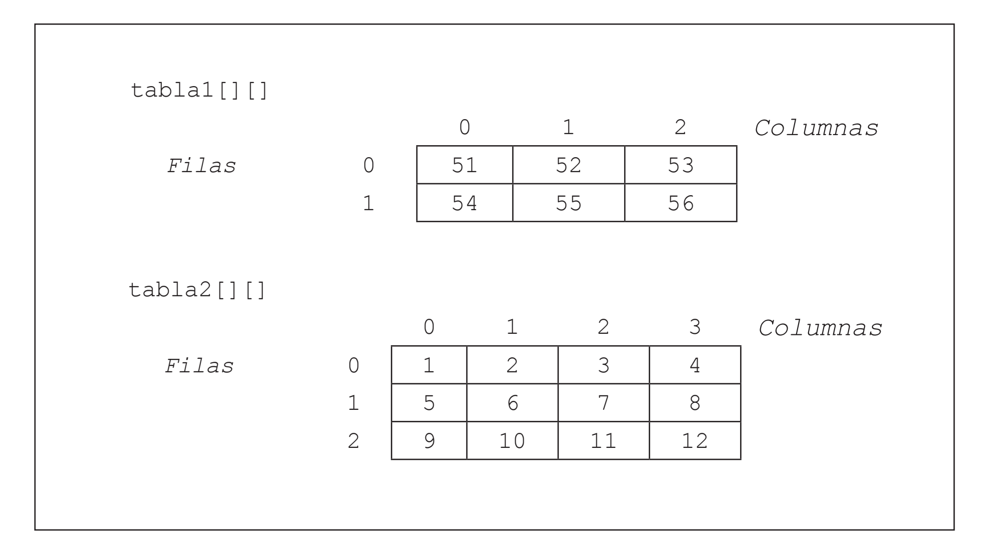


Figura 3.3 Tablas de dos dimensiones

Java trata los *arrays* de dos o más dimensiones como *arrays* de *arrays,* por esa razón se pueden crear *arrays* de dos dimensiones no cuadradas.

Ejemplo 3.6

Declaración y creación de arrays bidimensionales de distinto número de elementos por fila.

1.
```
double tb[][] = { {1.5, -2.5}, {5.0, -0.0, 1.5} };
```
Se ha definido una matriz de `2` filas, la primera con dos columnas y la segunda con `3`.

2.
```
int[] a = {1,3,5};
int [] b = {2,4,6,8,10};
int mtb[][] = {a, b};
```
Se ha definido el *array* `a` de `3` elementos, el `b` de `4` elementos y la matriz `mtb` de `2` filas, la primera con `3` elementos o columnas y la segunda con `4`.

Java permite crear matrices de distintas formas, la definición que se realiza en el Ejemplo 3.7 especifica primero el número de filas y, a continuación, el número de elementos de cada fila.

Ejemplo 3.7

Creación de arrays bidimensionales de distinto número de elementos por fila. Primero se crean las filas y, después, las columnas de cada fila.

1.
```
double [][]gr = new double[3][];
```
Define la matriz `gr` de `3` filas. A continuación los elementos de cada fila:
```
gr[0] = new double[3];
gr[1] = new double[6];
gr[2] = new double[5];
```

2.
```
int [][]pres = new int[4][];
```
Define la matriz `pres` de tipo entero con `4` filas. A continuación, los elementos de cada fila se definen con sentencias de inicialización:
```
pres[0] = {1,3,5,7};
pres[1] = {2,6,8};
pres[2] = {9,11};
pres[4] = {10};
```

Nota

En un array bidimensional `tabla`, al ser un array de arrays, el atributo `length` de `tabla` contiene el número de filas. El atributo `length` de cada array fila contiene el número de columnas.

```
float ventas[][] = {{0.,0.,0.},{1.0.,1.0},{-1.0}};
System.out.print(ventas.length);    // escribe 3
System.out.print(ventas[0].length); // escribe 3
System.out.print(ventas[1].length); // escribe 2
System.out.print(ventas[2].length); // escribe 1
```

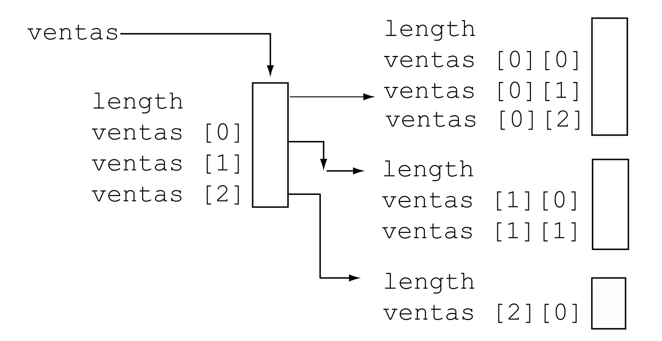


Figura 3.4 Disposición en memoria de `ventas [][]`

> ***Precaución***
>
> En la definición de un array bidimensional no es posible omitir el número de filas. Así, la declaración: `double vt[][] = new double[][4];` es errónea ya que no se ha especificado el número de filas, y el tamaño queda indeterminado.

3.2.2. Acceso a los elementos de *arrays* bidimensionales

El acceso a los elementos de *arrays* bidimensionales sigue el mismo formato que el acceso a los elementos de un *array* unidimensional. En este caso, con las matrices deben especificarse los índices de la fila y la columna.

El formato general para la asignación directa de valores a los elementos es:

```
inserción de elementos
  <nombre array>[indice fila][indice columna] = valor elemento;

extracción de elementos
  <variable> = <nombre array> [indice fila][indice columna];
```

Con dos bucles anidados se accede a todos los elementos de una matriz. Su sintaxis es:

```
int fila, col;
for (fila = 0; fila < NumFilas; ++fila)
   for (col = 0; col < NumCol; ++col)
      Procesar elemento Matriz[fila][col];
```

El número de filas y de columnas se puede obtener con el atributo `length`. Con este atributo, la sintaxis para acceder a los elementos es:

```
<tipo> Matriz[][] ;
<especificación de filas y columnas con operador new>
for (fila = 0; fila < Matriz.length; ++fila)
   for (col = 0; col < Matriz[fila].length; ++col)
      Procesar elemento Matriz[fila][col];
```

Ejercicio 3.3

Codificar un programa para dar entrada y posterior visualización de un array de dos dimensiones.

El método `leer()` da entrada a los elementos de la matriz que se pasan como argumento, y el método `visualizar()` muestra la tabla en la pantalla.

```
import java.io.*;
class tabla
{
   public static void main(String [] a) throws Exception
   {
      int v[][]= new int[3][5];
      leer(v);
      visualizar(v);
   }
   static void leer(int a[][])throws Exception
   {
      int i,j;
      BufferedReader entrada = new BufferedReader
                              (InputStreamReader(System.in));
      System.out.println("Entrada de datos de la matriz");
      for (i = 0; i < a.length; i++)
      {
         System.out.println("Fila: " + i);
         for (j = 0; j < a[i].length; j++)
            a[i][j]= Integer.parseInt(entrada.readLine());
      }
   }
   static void visualizar (int a[][])
   {
      int i,j;
      System.out.println("\n\t Matriz leida\n");
      for (i = 0; i < a.length; i++)
      {
         for (j = 0; j < a[i].length; j++)
            System.out.print(a[i][j] + " ");
         System.out.println(" ");
      }
   }
}
```

3.2.3. *Arrays* de más de dos dimensiones

Java proporciona la posibilidad de almacenar varias dimensiones, aunque raramente los datos del mundo real requieren más de dos o tres dimensiones. El medio más fácil de dibujar un *array* de tres dimensiones es imaginar un cubo, tal como se muestra en la Figura 3.5. Un *array* tridimensional se puede considerar como un conjunto de *arrays* bidimensionales combinados para formar, en profundidad, una tercera dimensión. El cubo se construye con filas (dimensión vertical), columnas (dimensión horizontal) y planos (dimensión en profundidad). Por consiguiente,

un elemento dado se localiza especificando su plano, fila y columna. A continuación se declara y define un *array* tridimensional `equipos`:

```
int equipos[][][]= new int[3][15][10];
```

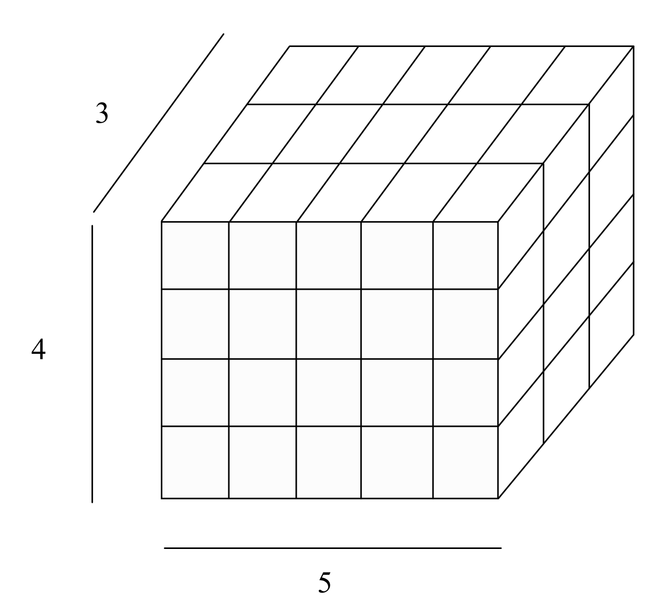


Figura 3.5 Un *array* de tres dimensiones (4 x 5 x 3)

Ejemplo 3.8

Crear un array tridimensional para representar los caracteres de un libro y diseñar los bucles de acceso.

El *array* `libro` tiene tres dimensiones, `[PAGINAS] [LINEAS] [COLUMNAS]`, que definen el tamaño del *array*. El tipo de datos del *array* es `char`, ya que los elementos son caracteres.

El método más fácil para acceder a los caracteres es mediante bucles anidados. Dado que el libro se compone de un conjunto de páginas, el bucle más externo es el bucle de página, y el bucle de columnas es el bucle más interno. Esto significa que el bucle de filas se insertará entre los bucles de página y de columna.

```
int pagina, linea, columna;
final int PAGINAS = 500;
final int LINEAS = 45;
final int COLUMNAS = 80;
char libro[][][] = new char[PAGINAS][ LINEAS][COLUMNAS];
for (pagina = 0; pagina < PAGINAS; ++pagina)
   for (linea = 0; linea < LINEAS; ++linea)
      for (columna = 0; columna < COLUMNAS; ++columna)
         <procesar libro[pagina][linea][columna]>
```

3.3. UTILIZACIÓN DE *ARRAYS* COMO PARÁMETROS

En Java, todas las variables de tipos primitivos *(double, float, char, int, boolean)* se pasan por valor. Por contra, los objetos siempre se pasan por referencia, y como los *arrays* son objetos, también se pasan por referencia (dirección). Esto significa que cuando se llama a un método y se utiliza un *array* como parámetro, se puede modificar el contenido de los elementos del *array* en el método. La Figura 3.6 ayuda a comprender el mecanismo.

```
public static void
main(String [] a)
{
   char palabra[]={'A','B','C','D'};
   cambiar(palabra);
}
```

palabra

```
void
cambiar(char c[])
{
  c[2] = 'c';
}
```

Figura 3.6 Paso de un *array* por dirección

El argumento del método que se corresponde con un *array* se declara poniendo el tipo de los elementos, el identificador y los corchetes en blanco, tantos corchetes como dimensiones. El número de elementos del *array* no se pasa como argumento, ya que el tamaño del *array* se conoce con el atributo `length`. El método `SumaDeMats()` tiene dos argumentos *array* bidimensional:

```
void SumaDeMats(double m1[][], double m2[][]);
```

Al método `SumaDeMats()` se pueden pasar dos argumentos de tipo *array*. Por ejemplo:

```
final int N = 5;
double mat1[][] = new double[N][N];
double mat2[][] = new double[N][N];

SumaDeMats(mat1,mat2);
```

Ejercicio 3.4

Paso de arrays a métodos. Se lee un array y se escribe el producto de los elementos positivos.

El número de elementos del *array* se establece en la ejecución del programa. Al método `leerArray()` se le pasa el *array* para dar entrada a sus valores. Al método `producto()` también se le pasa el *array;* devuelve el producto de los elementos positivos.

```
import java.io.*;
class ProductoMat
{
```

```
   static BufferedReader entrada = new BufferedReader(
                              new InputStreamReader(System.in));

   public static void main(String [] a)throws Exception
   {
      double v[];
      int n;
      System.out.print("Número de elementos: ");
      n = Integer.parseInt(entrada.readLine());
      v = new double[n];
      leerArray(v);
      System.out.println("El producto de los elementos= " +
                                                   producto(v));
   }
   static void leerArray(double a[])throws Exception
   {
      int n = 0;
      System.out.println("Introduzca " + a.length + "datos.");
      for (; n < a.length; n++)
      {
         a[n] = Double.valueOf(entrada.readLine()).doubleValue();
      }
   }
   static double producto(double w[])
   {
      double pd = 1.0;
      int n = w.length - 1;
      while (n > 0)
         if (w[n] > 0.0)
            pd *= w[n--];
         else
            n--;
      return pd;
   }
}
```

3.3.1. PRECAUCIONES

Un método conoce cuántos elementos existen en el *array* pasado como argumento. Puede ocurrir que no todos los elementos sean significativos, si esto ocurre hay que pasar un segundo argumento que indique el número real de elementos.

Ejemplo 3.9

El método `SumaDeEnteros()` suma los valores de los n elementos de un array y devuelve la suma.

```
int SumaDeEnteros(int[]arrayEnteros, int n)
{
   int i, s;
   for (i = s = 0; i < n; )
      s += arrayEnteros[i++];
   return s;
}
```

Aunque `SumaDeEnteros()` conoce la capacidad del *array* a través del atributo `length`, no sabe cuántos elementos hay que sumar y por ello se le pasa el parámetro `n` con el número verdadero de elementos. Una posible llamada al método es la siguiente:

```
int lista[] = new int[33];
n = 10;
SumaDeEnteros(lista, n);
```

Nota

Se pueden utilizar dos formas alternativas para permitir que un método conozca el número de elementos asociados con un array que se pasa como argumento al método:

- situar un valor de señal al final del array, que indique al método que se ha de detener el proceso en ese momento;
- pasar un segundo argumento que indica el número de elementos del array.

Ejercicio 3.5

Se lee una lista de, como máximo, 21 números enteros; a continuación, se calcula su suma y el valor máximo. La entrada de datos termina al introducir la clave -1.

El programa consta del método `entrada()`, que lee desde el teclado los elementos del *array* hasta que se lee el dato clave y devuelve el número de elementos leídos que nunca puede ser mayor que el máximo de elementos (atributo `length`). El método `sumaEnteros()` calcula la suma de los elementos introducidos en el *array* y se pasan dos parámetros, el *array* y el número de elementos. El método `maximo()` tiene los mismos parámetros que `sumaEnteros()`, determina el valor máximo.

```
import java.io.*;
class SumaMax
{
   public static void main(String [] a)throws Exception
   {
      final int NUM = 21;
      int items[] = new int[NUM];
      int n;
      n = entrada(items);  // devuelve el número real de elementos
      System.out.println("\nSuma de los elementos: " +
                                            sumaEnteros(items,n));
      System.out.println("\nValor máximo: " + maximo(items,n));
   }
   static int entrada(int w[])throws Exception
   {
      int k = 0, x;
      BufferedReader entrada = new BufferedReader(
                                 new InputStreamReader(System.in));
```

```
      System.out.println("Introduzca un máximo de " + w.length +
                         "datos, terminar con -1");
      do {
         x = Integer.parseInt(entrada.readLine());
         if (x != -1 )
         w[k++] = x;
      }while ((k < w.length) && (x != -1));
      return k;
   }
   static int sumaEnteros(int w [], int n)
   {
      int i, total = 0;
      for (i = 0; i < n; i++)
         total += w[i];
      return total;
   }
   static int maximo(int w[], int n)
   {
      int mx, i;
      mx = w[0];
      for (i = 1; i < n; i++)
         mx = (w[i]>mx ? w[i]: mx);

      return mx;
   }
}
```

3.4. CADENAS. CLASE `String`

Una *cadena* es una secuencia de caracteres delimitada entre dobles comillas, `"AMIGOS"` es una cadena (también llamada *constante de cadena* o *literal de cadena*) compuesta de 6 elementos `char (unicode)`.

Java no tiene el tipo cadena como tipo de datos primitivo sino que declara varias clases para el manejo de cadenas, de las que la más importante es la clase `String`. Cualquier cadena es considerada un objeto `String`. Por ejemplo:

```
String mipueblo = "Lupiana";
String vacia = "";
String rotulo = "\n\t Lista de pasajeros\n";
```

En Java, una cadena es un objeto tipo `String`, muy distinto de un array de caracteres.

Ejemplo 3.10

Se declaran *arrays* de caracteres y variables `String`

1. `char cad[] = {'L', 'u', 'p', 'i', 'a', 'n', 'a'};`

   `cad` es un *array* de siete caracteres.

2. El *array* de caracteres `cad` no conviene escribirlo directamente, pueden salir caracteres extraños si hay posiciones no asignadas; es conveniente transformar el *array* en una cadena.

```
System.out.println(cad);
```

El sistema no permite escribir en pantalla un *array,* a no ser elemento a elemento.

```
String pd = "Felicidad";
System.out.println(pd);
```

imprime en la pantalla la cadena `pd`.

3. `String cde = entrada.readLine();`

El método `readLine()` lee caracteres hasta fin de línea y devuelve una cadena formada por los caracteres captados, que es copiada en el objeto `cde` .

3.4.1. Declaración de variables Cadena

Las cadenas se declaran como cualquier objeto de una clase. El tipo base es la clase `String`:

```
String ch, cad;
```

La declaración ha creado dos referencias, `ch` y `cad`. Para crear el objeto se aplica el operador `new`, o bien se inicializa a una constante cadena.

> Una declaración como ésta: `String s` es simplemente una declaración que todavía no referencia a un objeto cadena .

3.4.2. Inicialización de variables Cadena

Las variables cadena, objetos cadena, se crean con el operador `new`, o bien con una sentencia de inicialización a una cadena constante.

```
String texto = "Esto es una cadena";
String textodemo = "Esta es una cadena muy larga";
String cadenatest = "¿Cuál es la longitud de esta cadena?";
```

Las variables `texto`, `textodemo` y `cadenatest` referencian las cadenas asignadas, se ha reservado memoria en el momento de la inicialización; se dice que son cadenas inmutables en el sentido de que no pueden cambiar el contenido, ni expandirse, ni eliminar caracteres. Estas variables son referencias y, como consecuencia, pueden cambiar esa referencia a la de otra cadena. Por ejemplo:

```
System.out.println("Cadena: " + cadenatest + cadenatest.length());
```

Si en una sentencia posterior se hace la asignación y salida por pantalla:

```
cadenatest = "ABC";
System.out.println("Cadena: " + cadenatest + cadenatest.length());
```

Ocurre que `cadenatest` referencia la constante `"ABC"` y la otra cadena ha dejado de ser referenciada; en ese momento, es candidata a que sea liberada la memoria que ocupa por el *recolector de memoria*.

3.4.3. Inicialización con un constructor de String

La inicialización de una variable cadena no sólo se puede hacer con un literal o cadena constante, sino también con alguno de los constructores de la clase `String` y el operador `new`. A continuación se describen los constructores mas útiles.

1. **Constructor de cadena vacía.**

   Crea una cadena sin caracteres, es una referencia a una cadena vacía. Si a continuación se obtiene su longitud (método `length()`), será cero.

```
String c;
c = new String();
```

2. **Constructor de cadena a partir de otra cadena.**

   Con este constructor se crea un objeto cadena a partir de otro objeto cadena ya creado.

```
String c1,c2;
. . .
c2 = new String(c1);

String mcd;
mcd = new String("Cadena constante");
```

3. **Constructor de cadena a partir de un objeto `StringBuffer`.**

   Los objetos de la clase `StringBuffer` tienen la particularidad de que son modificables, mientras que los objetos `String` no pueden cambiarse una vez creados. Con este constructor, una cadena `(String)` se crear a partir de otro objeto `StringBuffer`. Por ejemplo:

```
String cc;
StringBuffer bf = new StringBufffer("La Alcarria");
cc = new String(bf);
```

4. **Constructor de cadena a partir de un *array* de caracteres.**

   La cadena se inicializa con los elementos de un *array* de caracteres pasando como argumento al constructor.

```
String cdc;
char vc[]
. . .
cdc = new String(vc);
```

Ejemplo 3.11

`cad1` *referencia un objeto cadena creado a partir de un literal; a continuación se crea con el constructor que tiene como argumento la cadena* `cad1` *otro objeto cadena. La comparación que se hace con el método* `equals()` *es true.*

```
String cad1 = "Sabado tarde", cad2;
cad2 = new String(cad1);

if (cad1.equals(cad2))          // esta condición es true
System.out. println(cad1 + " = " + cad2);
```

Si la comparación se hubiera hecho con el operador == el resultado sería *false*. Esto es debido a que el operador compara las referencias y éstas son distintas; el método `equals()` compara el contenido de las cadenas y éstas son iguales.

Ejemplo 3.12

Un array de caracteres contiene las letras mayúsculas del alfabeto. La cadena se crea pasando al constructor el array de caracteres.

```
String mayus;
char []vmay = new char[26];
for (int j = 0; j < 26; j++)
   vmay[j] = (char)'A' + j;
mayus = new String(vmay);
```

3.4.4. Asignación de cadenas

Las cadenas son objetos, por consiguiente las variables de tipo `String` son referencias a esos objetos. La asignación de variables `String` no copian los objetos (la cadena) sino las referencias. Por ejemplo:

```
String c1d = new String("Buenos días");
String c2d = new String("Buenos días");

boolean sw;
sw = c1d == c2d;
```

`sw` toma el valor *false*, las referencias son distintas.

```
c1d = c2d;
```

a partir de esta asignación, `c1d` y `c2d` referencian el mismo objeto cadena. El anterior objeto referenciado por `c2d` será liberado.

```
sw = c1d == c2d;
```

`sw` toma el valor *true,* las dos variables referencian al mismo objeto.

Ejemplo 3.13

Una variable cadena se inicializa a un literal. Se realizan comparaciones que muestran las diferencias entre el operador == y el método `equals()` *.*

```
String nombre;
nombre = "Mariano";
// nombre referencia al objeto cadena "Mariano"
if (nombre == "Mariano" ) // compara referencias con operador ==
   System.out.println(nombre + " == Mariano : true" );
else
   System.out.println(nombre + " == Mariano : false" );
if (nombre.equals("Mariano" ) //compara contenidos
   System.out.println("Los objetos contienen lo mismo.");
else
   System.out.println("Los objetos son diferentes."");
```

La primera condición, `nombre == "Mariano"`, es *true* porque previamente se ha asignado a `nombre` la referencia de `"Mariano"`. La segunda condición, evaluada con el método `equals()`, se puede pensar de inmediato que también es *true*.

3.4.5. Métodos de `String`

La clase `String` dispone de un amplio conjunto de métodos para realizar operaciones con cadenas. Conviene recordar que una vez creada, la cadena de tipo `String` no puede modificarse, por eso, generalmente, estos métodos devuelven otra referencia con la nueva cadena. La Tabla 3.2 contiene un resumen de los métodos de `String`.

Tabla 3.2 Métodos de la clase `String`

Método	Cabecera del método y funcionalidad
length	`int length();` Devuelve el número de caracteres.
concat	`String concat(String arg2);` Añade la cadena `arg2` al final de cadena *invocante, concatena*.
charAt	`char charAt(int posicion);` Devuelve el carácter cuyo índice es `posicion`.
getChars	`void getChars(int p1, int p2, char[]ar, int inicial);` Obtiene el rango de caracteres comprendidos entre `p1` y `p2`, y los copia en `ar` a partir del índice `inicial`.
substring	`String substring(int inicial, int final);` Devuelve una cadena formada por los caracteres entre `inicial` y `final`.
compareTo	`int compareTo(String);` Compara alfabéticamente dos cadenas, la cadena invocante `(c1)` y la que se pasa como argumento `(c2)`. Devuelve: `= 0` si son iguales. `< 0` si alfabéticamente es menor `c1` que `c2`. `> 0` si alfabéticamente es mayor `c1` que `c2`.
equals	`boolean equals(String cad2);` Devuelve `true` si la cadena que llama coincide alfabéticamente con `cad2` (tiene en cuenta mayúsculas y minúsculas).
equalsIgnoreCase	`boolean equalsIgnoreCase(String cad2);` Devuelve `true` si la cadena que llama coincide alfabéticamente con `cad2` (sin tener en cuenta mayúsculas y minúsculas).
startsWith	`boolean startsWith(String cr);` `boolean startsWith(String cr, int posicion);` Compara desde el inicio de la cadena que llama, o bien a partir de `posicion`, con la cadena `cr`.
endsWith	`boolean endsWith(String cad2);` Compara desde el final de la cadena que llama con `cad2`.

(*Continúa*)

Método	Cabecera del método y funcionalidad
regionMatches	`boolean regionMatches(boolean tip, int p1, String cad2, int p2, int nc);` Compara `nc` caracteres tanto de la cadena que llama como de la cadena `cad2`, a partir de las posiciones `p1 y p2` respectivamente. Según esté a `true tip`, o no tiene en cuenta mayúsculas y minúsculas.
toUpperCase	`String toUpperCase();` Convierte la cadena en otra cadena con todas las letras en mayúsculas.
toLowerCase	`String toLowerCase();` Convierte la cadena en otra cadena con todas las letras en minúsculas.
replace	`String replace(char c1, char c2);` Sustituye todas las ocurrencias del carácter `c1` por el carácter `c2`, devuelve la nueva cadena.
trim	`String trim();` Elimina los espacios, tabuladores o caracteres de fin de línea de inicio a final de la cadena.
toCharArray	`char[]toCharArray();` Devuelve los caracteres de la cadena como un *array* de caracteres.
valueOf	`String valueOf(tipo_dato_primitivo);` Convierte cualquier dato perteneciente a los tipos primitivos en una cadena.
indexOf	`int indexOf(int c);` `int indexOf(int c, int p);` `int indexOf(String b, int p);` Busca un carácter o bien otra cadena desde la posición `0`, o desde la posición `p`.
lastIndexOf	`int lastIndexOf(int c);` `int lastIndexOf(int c, int p);` `int lastIndexOf(String b, int p);` Busca un carácter o bien otra cadena desde la posición `length()-1`, o desde la posición `p`, desde el final de la cadena al principio.

3.4.6. Operador + con cadenas

El operador + aplicado a cadenas da como resultado otro objeto cadena que es la unión o concatenación de ambas. Por ejemplo:

```
String c1 = "Ángela";
String c2 = "Paloma";
String c3 = c1 + c2; // genera una nueva cadena: AngelaPaloma
String cd;
cd = "Musica" + "clasica"; // genera la cadena Musicaclasica
```

También se puede concatenar una cadena con un tipo primitivo. Por ejemplo,

```
String c;
c = 34 + " Cartagena";      // genera la cadena "34 Cartagena"
c = c + " Madrid" + '#';   // genera la cadena "34 Cartagena Madrid#"
```

Se puede aplicar el operador + para concatenar una cadena con cualquier otro dato. ¿Cómo se produce la concatenación? Existe el método `toString()` que, primero, convierte el dato en una representación en forma de cadena de caracteres y a continuación se unen las cadenas.

Ejemplo 3.14

Concatenación de cadenas con distintos datos.

```
String ch;
ch = new String("Patatas a ");
double x = 11.2;
ch = ch + x + " Euros"; // genera la cadena: "Patatas a 11.2 Euros"

String bg;
bg = 2 + 4 + "Mares";   // genera la cadena "6Mares", primero suma 2+4
                        // y a continuación concatena.
bg = 2 + (4 + "Mares"); // genera la cadena "24Mares", los
                        // paréntesis cambian el orden de evaluación.
bg = "Mares" + 2 + 4;   /* genera la cadena "Mares24", primero
                        concatena "Mares"+2 dando lugar a "Mares2"; a
                        continuación concatena "Mares2" con 4 */
```

3.5. CLASE `Vector`

Java proporciona un grupo de clases que almacenan secuencias de objetos de cualquier tipo, son las *colecciones*. Se diferencian en la forma de organizar los objetos y, en consecuencia, la manera de recuperarlos. La clase `Vector` (paquete `java.util`) es una de estas *colecciones,* tiene un comportamiento similar a un *array* unidimensional.

Un `Vector` guarda objetos (referencias) de cualquier tipo y crece dinámicamente, sin necesidad de tener que programar operaciones adicionales. El *array* donde almacena los elementos es de tipo `Object`. Su declaración:

```
protected Object elementData[]
```

3.5.1. Creación de un Vector

Se utiliza el operando `new` de igual forma que para crear cualquier objeto. La clase `Vector` dispone de diversos constructores:

`public Vector()`	*crea un vector vacío.*
`public Vector(int capacidad)`	*crea un vector con una capacidad inicial.*
`public Vector(Collection org)`	*crea un vector con los elementos de* `org`.

Por ejemplo:

```
Vector v1 = new Vector();
Vector v2 = new Vector(100);
Vector v3 = new Vector(v2); // v3 contiene los mismo elementos que v2
```

3.5.2. Insertar elementos

La clase dispone de diferentes métodos para insertar o añadir elementos al vector. Los elementos que se meten en el vector deben ser objetos, no pueden ser datos de tipos primitivos (`int`, `char` ...).

`boolean add (Object ob);`	añade el objeto a continuación del último elemento del vector.
`void addElement(Object ob);`	añade el objeto a continuación del último elemento del vector.
`void insertElement (Object ob, int p);`	inserta el objeto en la posición `p`; los elementos posteriores a `p` se desplazan.

3.5.3. Acceso a un elemento

Se accede a un elemento del vector por la posición que ocupa. Los métodos de acceso devuelven el elemento con el tipo `Object`, por esa razón es posible que sea necesario realizar una conversión al tipo del objeto.

`Object elementAt(int p);`	devuelve el elemento cuya posición es `p`.
`Object getElement(int p);`	devuelve el elemento cuya posición es `p`.
`Object get(int p);`	devuelve el elemento cuya posición es `p`.
`int size();`	devuelve el número de elementos.

3.5.4. Eliminar un elemento

Un vector es una estructura dinámica, crece o decrece según se añaden o se eliminan objetos. Se puede eliminar un elemento de distintas formas, por ejemplo, por la posición que ocupa (índice); a partir de esa posición, el resto de elementos del vector se mueven una posición a la izquierda y disminuye el número de elementos. Otra forma de eliminar es transmitiendo el objeto que se desea retirar del vector. También hay métodos de la clase para eliminar todos los elementos que son iguales a una *colección;* incluso se pueden eliminar todos los elementos.

`void removeElementAt(int indice);`	elimina elemento `índice` y el resto se renumera.
`boolean void removeElement (Object op);`	elimina la primera aparición de `op`; devuelve `true` si realiza la eliminación.
`void removeAll(Collection gr);`	elimina los elementos que están en `gr`.
`void removeAllElements();`	elimina todos los elementos.

3.5.5. Búsqueda

Los diversos métodos de búsqueda de `Vector` devuelven la posición de la primera ocurrencia del objeto buscado, o bien *verdadero-falso* según el éxito de la búsqueda.

`boolean contains(Object op);`	devuelve `true` si encuentra `op`.
`int indexOf(Object op);`	devuelve la primera posición de `op`, `-1` si no está.

Ejercicio 3.6

Utilizar un vector para guardar indistintamente, números racionales y números complejos.

Se supone que un número racional está representado por dos enteros, numerador y denominador respectivamente. Un número complejo también viene representado por dos enteros, parte real y parte imaginaria. Entonces, se declara la clase `Numero` con los atributos de tipo entero `x`, `y`; las clases `Racional` y `Complejo` derivan de `Numero`. Además, el método `mostrar()` se redefine en cada clase para escribir el tipo de número. La clase principal crea un vector al que se añaden números racionales y complejos alternativamente. A continuación se recuperan los elementos y se escriben.

```
import java.util.*;
import java.io.*;

abstract class Numero
{
   protected int x, y;
   public Numero() {;}
   public Numero(int _ x, int _ y)
   {
      x = _ x;
      y = _ y;
   }
   abstract void mostrar();
}

class Racional extends Numero
{
   public Racional() {;}
   public Racional(int _ x, int _ y) { super( _ x, _ y); }
   void mostrar()
   {
      System.out.println(x + "/" + y);
   }
}

class Complejo extends Numero
{
   public Complejo() {;}
   public Complejo(int _ x, int _ y) { super( _ x, _ y); }
   void mostrar()
   {
      System.out.println("(" + x + "," + y + ")");
   }
}
```

```
public class VectorNumero
{
   static final int N = 10;
   public static void main (String [] a)
   {
      Vector num = new Vector();
      for(int i = 1; i <= N; i++)
      {
         Numero q;
         q = new Racional(3 * i, 3 * i % 7 + 1);
         num.addElement(q);
         q = new Complejo(3 * i % 7, 3 * i - 5);
         num.addElement(q);
      }
      // recuperación de los elementos
      int k;
      k = num.size();            // número de elementos
      for (int i = 1; i <= N; i++)
      {
         Numero q;
         q = (Numero) num.elementAt(i);
         q.mostrar();
      }
   }
}
```

RESUMEN

En este capítulo se analizan los tipos agregados de Java, *arrays* y diversos métodos para el manejo de cadenas incluidos en la clase `String`, `StringBuffer`. Después de leer este capítulo, debe tener un buen conocimiento de los conceptos fundamentales de los tipos agregados.

Se describen y analizan los siguientes conceptos:

- Un *array* es un tipo de dato estructurado que se utiliza para localizar y almacenar elementos de un tipo de dato dado.
- Existen *arrays* de una dimensión, de dos dimensiones, y multidimensionales.
- En Java, los *arrays* se declaran especificando el tipo de dato del elemento, el nombre del *array* y tantos corchetes como dimensiones. El tamaño de cada dimensión del *array* se define con el operador `new`. Para acceder a los elementos del *array* se deben utilizar sentencias de asignación directas, sentencias de lectura/escritura o bucles (mediante las sentencias `for`, `while` o `do-while`).

```
int total _ meses[] = new int[12];
```

- Los *arrays* en Java son considerados objetos. Por ello, todos tienen el atributo `length` con la longitud del *array*.

- Los arrays de caracteres en Java son secuencias de caracteres individuales. Java tiene la clase `String` para el tratamiento de cadenas.
- Las cadenas en Java son objetos de la clase `String`, una vez creados no pueden ser modificados; las cadenas del tipo `StringBuffer` si pueden cambiar dinámicamente.
- La clase `String` dispone de constructores para inicializar las cadenas y numerosos métodos de manipulación; destacan los métodos que soportan concatenación, conversión, inversión y búsqueda.
- Java considera las cadenas objetos; las variables `String` contienen la referencia al objeto cadena. La asignación de una variable a otra supone asignar la referencia al objeto.

Además, en este capítulo se estudia la clase `Vector`, que se encuentra en el paquete `java.util`. Los objetos `Vector` permiten guardar y recuperar objetos de cualquier tipo, se comportan como arrays de tipo `Object`.

EJERCICIOS

Para los ejercicios del 3.1 a 3.6 suponga las declaraciones:

```
int i,j,k;
int Primero[] = new int[21];
int Segundo[] = new int[21];
int Tercero[][] = new int[7][8];
BufferedReader entrada = new BufferedReader(
                                 (new InputStreamReader(System.in));
```

Determinar la salida de cada segmento de programa (en los casos que se necesite, se indica debajo el archivo de datos de entrada correspondiente; con `ctrl r` se indica que hay un fin de línea.).

3.1.

```
for (i=1; i <= 6; i++)
  Primero[i] = Integer.parseInt(entrada.readLine());
for (i=3; i>0; i--)
  System.out.print(Primero[2*i] + " ");
.................
3 ctrl+ r 7 ctrl+ r 4 ctrl+ r -1 ctrl+ r 0 ctrl+ r 6
```

3.2.

```
k = Integer.parseInt(entrada.readLine());
for (i=3; i<=k;)
  Segundo[i++] = Integer.parseInt(entrada.readLine());
j= 4;
System.out.println( + Segundo[k] + " " + Segundo[j+1]);
............
6 ctrl+ r 3 ctrl+ r 0 ctrl+ r 1 ctrl+ r 9
```

3.3.
```
for (i= 0; i<10;i++)
  Primero[i] = i + 3;
j = Integer.parseInt(entrada.readLine());
k = Integer.parseInt(entrada.readLine());
for (i= j; i<=k;)
  System.out.println(Primero[i++]);
............
7 ctrl+ r 2 ctrl+ r 3 ctrl+ r 9
```

3.4.
```
for (i=0, i<12; i++)
  Primero[i] = Integer.parseInt(entrada.readLine());
for (j=0; j<6;j++)
  Segundo[j]=Primero[2*j] + j;
for (k=3; k<=7; k++)
  System.out.println(Primero[k+1] + " " + Segundo [k−1]);
...........
2 ctrl+ r 7 ctrl+ r 3 ctrl+ r 4 ctrl+ r 9 ctrl+ r -4 ctrl+ r
6 ctrl+ r -5 ctrl+ r 0 ctrl+ r 5 ctrl+ r -8 ctrl+ r 1
```

3.5.
```
for (j=0; j<7; )
  Primero[j++] = Integer.parseInt(entrada.readLine());
i = 0;
j = 1;
while ((j< 6) && (Primero[j−1]<Primero[j]))
{
   i++; j++;
}
for (k=-1; k<j+2; )
  System.out.println(Primero[++k]);
..........
20 ctrl+ r 60 ctrl+ r 70 ctrl+ r 10 ctrl+ r 0 ctrl+ r
40 ctrl+ r 30 ctrl+ r 90
```

3.6.
```
for (i=0; i<3; i++)
   for (j= 0; j<12; j++)
      Tercero[i][j] = i+j+1;
for (i= 0; i< 3; i++)
{
  j = 2;
  while (j < 12)
  {
    System.out.println(i + " " + j " " + Tercero[i][j]);
    j+=3;
  }
}
```

3.7. Escribir un programa que lea el *array:*

```
4    7    1    3    5
2    0    6    9    7
3    1    2    6    4
```

y lo escriba como:

```
4     2     3
7     0     1
1     6     2
3     9     6
5     7     4
```

3.8. Dado el array:

```
4     7     -5    4     9
0     3     -2    6     -2
1     2     4     1     1
6     1     0     3     -4
```

escribir un programa que encuentre la suma de todos los elementos que no pertenecen a la diagonal principal.

3.9. Escribir un método que intercambie la fila i-ésima por la j-ésima de un array de dos dimensiones, `mxn`.

3.10. Considerando el siguiente segmento de código, indicar los errores y la forma de corregirlos.

```
String b = "Descanso activo";
char p[] = "En la semana par.";
StringBuffer c = "Cadena dinámica";
b = cd;
StringBuffer fc = new Stringbuffer("Calor de verano");
b = fc;
```

3.11. Escribir un método que tenga como entrada una cadena y devuelva el número de vocales, de consonantes y de dígitos de la cadena.

3.12. ¿Qué diferencias y analogías existen entre las variables `c1, c2, c3`? La declaración es:

```
String c1;
String c2[];
StringBuffer c3;
```

3.13. Escribir un método que tenga como argumento una cadena con la fecha en formato: `dd/mm/aa` y devuelva una cadena con la fecha en formato `dd Mes(nominal) de año`. Por ejemplo:

```
21/4/01  debe transformarse a 21 Abril del 2001
```

3.14. Definir un array de cadenas para poder leer un texto compuesto por un máximo de 80 líneas. Escribir un método para leer el texto; el método debe de tener dos argumentos, uno, el texto y el segundo, el número de líneas.

PROBLEMAS

3.1. Un texto de n líneas tiene ciertos caracteres que se consideran comodines. Hay dos comodines, el # y el ?.

El primero indica que se ha de sustituir por la fecha actual, en formato `dia(nn) de Mes(nombre) año(aaaa)`, por ejemplo `21 de Abril 2001`. El otro comodín indica que se debe reemplazar por un nombre. Escribir un programa que lea las líneas del texto y cree un *array* de cadenas, de modo que cada elemento esté referenciando una cadena que es el resultado de realizar las sustituciones indicadas. La fecha y el nombre se han de obtener del flujo de entrada.

3.2. Escribir un programa que permita visualizar el triángulo de Pascal:

```
                  1
               1     1
            1     2     1
         1     3     3     1
      1     4     6     4     1
   1     5    10    10     5     1
1     6    15    20    15     6     1
```

En el triángulo de Pascal, cada número es la suma de los dos números situados encima de él. Este problema se debe resolver utilizando un *array* de una sola dimensión.

3.3. Se sabe que en las líneas que forman un texto hay valores numéricos enteros, que representan los kg de patatas recogidos en una finca. Los valores numéricos están separados de las palabras por un blanco, o por el carácter de fin de línea. Escribir un programa que lea el texto y obtenga la suma de los valores numéricos.

3.4. Escribir un programa que lea una cadena clave y un texto de, como máximo, 50 líneas. El programa debe de eliminar las líneas que contengan la clave.

3.5. Se desea sumar números grandes, tan grandes que no pueden almacenarse en variables de tipo `long`. Se ha pensado en introducir cada número como una cadena de caracteres y realizar la suma extrayendo los dígitos de ambas cadenas.

3.6. Escribir un programa que visualice un cuadrado mágico de orden impar *n* comprendido entre 3 y 11; el usuario debe elegir el valor de *n*. Un cuadrado mágico se compone de números enteros comprendidos entre 1 y *n*. La suma de los números que figuran en cada fila, columna y diagonal son iguales.

Ejemplo

8	1	6
3	5	7
4	9	2

Un método de generación consiste en situar el número 1 en el centro de la primera fila, el número siguiente en la casilla situada por encima y a la derecha, y así sucesivamente. El cuadrado es cíclico: la línea encima de la primera es, de hecho, la última y la columna a derecha de la última es la primera. En el caso de que el número generado caiga en una casilla ocupada, se elige la casilla situada encima del número que acaba de ser situado.

3.7. En el juego del ahorcado participan dos personas (o una persona y una computadora). Un jugador selecciona una palabra y el otro trata de adivinar la palabra acertando letras individuales. Diseñar un programa para jugar al ahorcado. *Sugerencia*: almacenar una lista de palabras en un `Vector` y seleccionar palabras aleatoriamente.

3.8. Escribir un programa que lea las dimensiones de una matriz, lea y visualice la matriz y a continuación, encuentre los elementos mayor y menor de la matriz y sus posiciones.

3.9. Si x representa la media de los números x_1, x_2,...x_n, entonces la *varianza* es la media de los cuadrados de las desviaciones de los números de la media.

$$Varianza = \frac{1}{n} \sum_{i=1}^{n} (x_i - x)^2$$

y la *desviación estándar* es la raíz cuadrada de la varianza. Escribir un programa que lea una lista de números reales, los cuente y, a continuación, calcule e imprima su media, varianza y desviación estándar. Utilizar un método para calcular la media, otro para calcular la varianza y otro para la desviación estándar.

3.10. Escribir un programa para leer una matriz A y formar la matriz transpuesta de A. El programa debe escribir ambas matrices.

3.11. Escribir un método que acepte como parámetro un *array* que puede contener números enteros duplicados. El método debe sustituir cada valor repetido por -5 y devolver el vector modificado y el número de entradas modificadas.

3.12. Los resultados de las últimas elecciones a alcalde en Carchelejo han sido los siguientes:

Distrito	*Candidato A*	*Candidato B*	*Candidato C*	*Candidato D*
1	194	48	206	45
2	180	20	320	16
3	221	90	140	20
4	432	50	821	14
5	820	61	946	18

Escribir un programa que haga las siguientes tareas:

a) Imprimir la tabla anterior con cabeceras incluidas.

b) Calcular e imprimir el número total de votos recibidos por cada candidato y el porcentaje del total de votos emitidos. Asimismo, visualizar el candidato más votado.

c) Si algún candidato recibe más del 50 por ciento de los votos, el programa imprimirá un mensaje declarándole ganador.

d) Si ningún candidato recibe más del 50 por ciento de los votos, el programa debe imprimir el nombre de los dos candidatos más votados, que serán los que pasen a la segunda ronda de las elecciones.

3.13. Una agencia de venta de automóviles distribuye quince modelos diferentes y tiene en su plantilla a diez vendedores. Se desea un programa que escriba un informe mensual de las ventas por vendedor y modelo, así como el número de automóviles vendidos por cada trabajador y el número total de cada modelo vendido por todos los vendedores. Asimismo, para entregar el premio al mejor vendedor, se necesita saber cuál es el empleado que más coches ha vendido.

vendedor \ modelo	1	2	3	...	15
1	4	8	1		4
2	12	4	25		14
3	15	3	4		7
...					
10					

3.14. Escribir un programa que lea una línea de caracteres y la visualice de tal forma que las vocales sean sustituidas por el carácter que más veces se repite en la línea.

3.15. Escribir un programa que encuentre dos cadenas introducidas por teclado que sean anagramas. Se considera que dos cadenas son anagramas si contienen exactamente los mismos caracteres en el mismo o en diferente orden. Hay que ignorar los blancos y considerar que las mayúsculas y las minúsculas son iguales.

3.16. Se dice que una matriz tiene un *punto de silla* si alguna posición de la matriz es el menor valor de su fila y, a la vez, el mayor de su columna. Escribir un programa que tenga como entrada una matriz de números reales y calcule la posición de un *punto de silla* (si es que existe).

3.17. Escribir un programa en el que se genere aleatoriamente un *array* de 20 números enteros. El *array* ha de quedar de tal forma que la suma de los 10 primeros elementos sea mayor que la suma de los 10 últimos elementos. Mostrar el *array* original y el *array* con la distribución indicada.

CAPITULO 4

Clases derivadas y polimorfismo

Objetivos

Con el estudio de este capítulo usted podrá:

- Conocer la propiedad de la orientación a objetos: herencia.
- Declarar una clase como derivada de otra clase.
- Encontrar las diferencias entre sobrecarga y redefinición de métodos.
- Especificar jerarquías de clases.
- Comprender el concepto de polimorfismo.
- Definir clases derivadas en Java.
- Crear objetos de clases derivadas.
- Definir clases abstractas en una jerarquía de clases.
- Realizar una pequeña aplicación de un problema resuelto con orientación a objetos.

Contenido

Conceptos clave

- ♦ Clase abstracta.
- ♦ Clase base.
- ♦ Clase derivada.
- ♦ Constructor.
- ♦ Declaración de acceso.
- ♦ Especificadores de acceso.
- ♦ Herencia.
- ♦ Herencia múltiple.
- ♦ Ligadura dinámica.
- ♦ Método abstracto.
- ♦ Relación *es-un*.
- ♦ Relación *tiene-un.*
- ♦ Polimorfismo.

Para profundizar (página web: www.mhe.es/joyanes)

- ▪ Ligadura dinámica mediante métodos abstractos.

INTRODUCCIÓN

En este capítulo se introduce el concepto de *herencia* y se muestra cómo crear *clases derivadas*. La herencia hace posible crear jerarquías de clases relacionadas y reduce la cantidad de código redundante en componentes de clases. El soporte de la herencia es una de las propiedades que diferencia los lenguajes *orientados a objetos* de los lenguajes *basados en objetos* y *lenguajes estructurados*.

La *herencia* es la propiedad que permite definir nuevas clases usando como base clases ya existentes. La nueva clase *(clase derivada)* hereda los atributos y el comportamiento que son específicos de ella. La herencia es una herramienta poderosa que proporciona un marco adecuado para producir software fiable, comprensible, de bajo coste, adaptable y reutilizable.

4.1. CLASES DERIVADAS

La *herencia* o relación **es-un** es la relación que existe entre dos clases: la clase denominada *derivada* que se crea a partir de otra ya existente, denominada *clase base*. La nueva clase *hereda* de la clase ya existente. Por ejemplo, si existe una clase `Figura` y se desea crear una clase `Triángulo`, esta clase puede derivarse de `Figura`, ya que tendrá en común con ella un estado y un comportamiento, aunque luego tendrá sus características propias. `Triángulo` *es-un* tipo de `Figura`. Otro ejemplo, puede ser *`Programador`* que *es-un* tipo de *`Empleado`*.

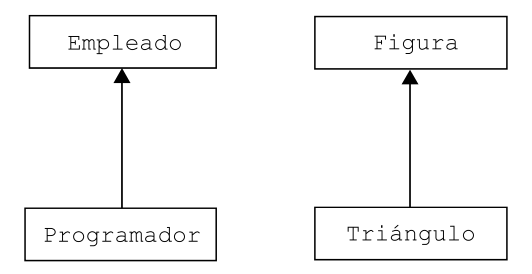


Figura 4.1 Clases derivadas

Evidentemente, la clase base y la clase derivada tienen código y datos comunes, de modo que si se crea la clase derivada de modo independiente, se duplicaría mucho de lo que ya se ha escrito para la clase base. Java soporta el mecanismo de *extensión* `(extends)` que permite crear clases derivadas o clases que son extensión de otra clase, de modo que la nueva clase *hereda* todos los miembros datos y los métodos que pertenecen a la clase ya existente.

La declaración de derivación de clases debe incluir la palabra reservada `extends` y, a continuación, el nombre de la clase base de la que se deriva. La primera línea de cada declaración debe incluir el formato siguiente:

```
class nombre _ clase extends nombre _ clase _ base
```

Regla

En Java se debe incluir la palabra reservada `extends` en la línea de la declaración de la clase derivada. Esta palabra reservada produce que todos los miembros no privados (`private`) de la clase base se hereden en la clase derivada.

Ejemplo 4.1

Declaración de las clases `Programador` *y* `Triangulo`.

1.
```
class Programador extends Empleado
{
 public miembro público
 // miembros públicos
 private miembro privado
 // miembros privados
}
```

2.
```
class Triangulo extends Figura
{
 public miembro público
   // miembros públicos
 protected miembro protegido
   // miembros protegidos
 ...
}
```

Una vez creada la clase derivada, el siguiente paso es añadir los nuevos miembros que se requieren para cumplir las necesidades específicas de la nueva clase.

clase derivada   clase base

```
class Director  extends Empleado
{
   public nuevo método
     ...
   private nuevo miembro
     ...

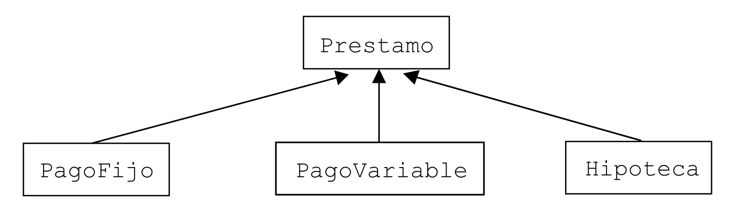
}
```

La declaración de la clase `Director` sólo tiene que especificar los nuevos miembros (métodos y datos). Todas los métodos miembro y los miembros dato de la clase `Empleado` (no privados) son heredados automáticamente por la clase `Director`. Por ejemplo, el método `calcular_salario()` de `Empleado` se aplica automáticamente a `Director`:

```
Director d;
d.calcular _ salario( );
```

Ejemplo 4.2

Considerar una clase `Prestamo` *y tres clases derivadas de ella:* `PagoFijo`, `PagoVariable` *e* `Hipoteca`.



La clase `Prestamo` es la clase base de la `PagoFijo`, `PagoVariable` e `Hipoteca`, se considera que es una clase abstracta, en ella se agrupan los métodos comunes a todo tipo de préstamo, hipoteca...

```
abstract class Prestamo
{
   final int MAXTERM = 22;
   protected float capital;
   protected float tasaInteres;
   public void prestamo(float p, float r) { ... };
   abstract public int crearTablaPagos(float mat[][]);
}
```

Las variables `capital` y `tasaInteres` no se repiten en la clase derivada.

```
class PagoFijo extends Prestamo
{
   private float pago; // cantidad mensual a pagar por cliente
   public PagoFijo (float x, float v, float t) { ... };
   public int crearTablaPagos(float mat [][]){ ... };
}

class PagoVariable extends Prestamo
{
   private float pago; // cantidad mensual a pagar por cliente
   public PagoVariable (float x, float v, float t) { ... };
   public int crearTablaPagos(float mat [][]){ ... };
}

class Hipoteca extends Prestamo
{
   private int numRecibos;
   private int recibosPorAnyo;
   private float pago;
   public Hipoteca(int a, int g, float x, float p, float r) { ... };
   public int crearTablaPagos(float mat [][]) { ... };
}
```

4.1.1. Declaración de una clase derivada

La sintaxis para la declaración de una clase derivada es la siguiente:

Nombre de la clase derivada — Palabra reservada para indicar derivación — Nombre de la clase base

```
class ClaseDerivada extends ClaseBase
{
     // miembros específicos de la clase derivada
}
```

La *clase base* (`ClaseBase`) es el nombre de la clase de la que deriva la nueva clase. Los miembros **`private`** de clase base son los únicos que la clase derivada no hereda, por lo que no se puede acceder a ellos desde métodos de clase derivada. Los miembros con visibilidad **`public`**, **`protected`** se incorporan a la clase derivada con la misma visibilidad que tienen en la clase base. Por ejemplo:

```
package personas;
public class Persona
{
   //miembros de la clase
}
```

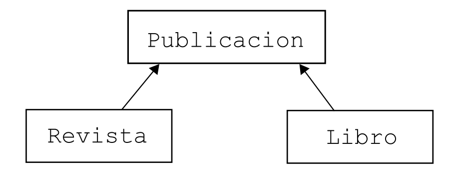
La clase `Persona` se puede utilizar en otros *paquetes* como clase base, o para crear objetos.

```
package empresa;
import personas.*;
public class Becario extends Persona
{
   //
}
```

Ejercicio 4.1

Representar la jerarquía de clases de publicaciones que se distribuyen en una librería: revistas, libros, etc.

Todas las publicaciones tienen en común la editorial y su fecha de publicación. Las revistas tienen una determinada periodicidad, el número de ejemplares que se publican al año, el número de ejemplares que se ponen en circulación controlados oficialmente (en España, por la OJD). Los libros tienen como características específicas el código ISBN y el nombre del autor.



```
class Publicacion
{
   public void nombrarEditor(String nomE){...}
   public void ponerFecha(long fe) {...}
   private String editor;
   private long fecha;
}
class Revista extends Publicacion
{
   public void fijarnumerosAnyo(int n) {...}
   public void fijarCirculacion(long n) {...}
   private int numerosPorAnyo;
   private long circulacion;
}
class Libro extends Publicacion
{
   public void ponerISBN(String nota) {...}
   public void ponerAutor(String nombre) {...}
   private String isbn;
   private String autor;
}
```

Con esta declaración, un objeto `Libro` contiene miembros datos y métodos heredados de la clase `Publicacion`, como el `isbn` y el nombre del `autor`. En consecuencia, serán posibles las siguientes operaciones:

```
Libro lib = new Libro();
lib.nombrarEditor("McGraw-Hill");
lib.ponerFecha(990606);
lib.ponerISBN("84-481-2015-9");
lib.ponerAutor("Mackoy, José Luis");
```

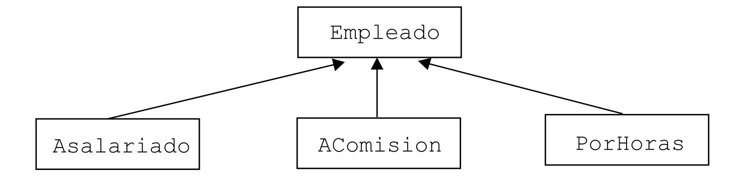
Para objetos de tipo `Revista`:

```
Revista rev = new Revista();
rev.fijarNumerosAnyo(12);
rev.fijarCirculacion(300000);
```

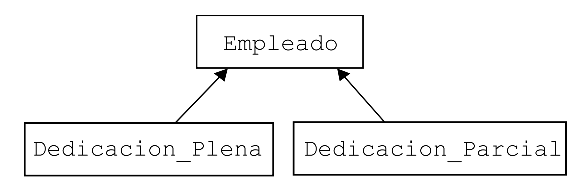
4.1.2. Diseño de clases derivadas

En el diseño de una aplicación orientada a objetos no siempre resulta fácil establecer la relación de herencia más óptima entre clases. Consideremos, por ejemplo, a los empleados de una empresa. Existen diferentes tipos de clasificaciones según el criterio de selección *(discriminador)*, que pueden ser: modo de pago (sueldo fijo, por horas, a comisión), dedicación a la empresa (plena o parcial) o estado de su relación laboral con la empresa (fijo o temporal).

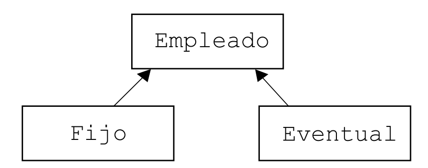
Una clasificación de los empleados basada en el modo de pago puede dividirlos en empleados con salario mensual fijo, empleados con pago por horas de trabajo y empleados con pago a comisión por las ventas realizadas.



Otra clasificación basada en la dedicación a la empresa: a *tiempo completo* o bien *por horas*.



Si el criterio de clasificación es la duración del contrato, los empleados se dividen en fijos o eventuales.



Una dificultad añadida a la que se enfrenta el diseñador es que un mismo objeto, en el supuesto anterior, un mismo empleado, puede pertenecer a diferentes grupos. Un empleado con dedicación plena puede ser remunerado con un salario mensual, un empleado con dedicación parcial puede ser remunerado mediante comisiones y un empleado fijo puede ser remunerado por horas. Una pregunta usual es: ¿cuál es la relación de herencia que describe la mayor cantidad de variación en los atributos de las clases y operaciones? ¿esta relación ha de ser el fundamento del diseño de clases? Evidentemente, la respuesta adecuada sólo se podrá dar cuando se tenga presente la aplicación real a desarrollar.

4.1.3. Sobrecarga de métodos en la clase derivada

La sobrecarga de métodos se produce cuando, al definir un método en una clase, tiene el mismo nombre que otro de la misma clase pero distinto número o tipo de argumentos. En la sobrecarga no interviene el tipo de retorno. La siguiente clase tiene métodos sobrecargados:

```
class Ventana
{
   public void copiar(Ventana w) {...}               // sobrecarga de
   public void copiar(String p, int x, int y) {...} // copiar()
}
```

Una clase derivada puede redefinir un método de la clase base sin tener exactamente la misma *signatura,* teniendo el mismo nombre pero distinta la lista de argumentos. Esta redefinición en la clase derivada no oculta al método de la clase base, sino que da lugar a una sobrecarga del método heredado en la clase derivada.

La clase `VentanaEspecial` derivada de `Ventana`, define el método `copiar()` con diferentes argumentos de los métodos `copiar()`; no se anulan los métodos, sino que están sobrecargados en la clase derivada.

```
class VentanaEspecial extends Ventana
{
   public void copiar(char c,int veces, int x, int y) {...}
   }
```

Ejemplo 4.3

Se declara una clase base con el método `escribe()` *y una clase derivada con el mismo nombre del método pero distintos argumentos. La clase con el método* `main()` *crea objetos y realiza llamadas a los métodos sobrecargados de la clase derivada.*

```
class BaseSobre
{
   public void escribe(int k)
   {
     System.out.print("Método clase base, argumento entero: ");
     System.out.println(k);
   }
   public void escribe(String a)
   {
     System.out.print("Método clase base, argumento cadena: ");
     System.out.println(a);
   }
}

class DerivSobre extends BaseSobre
{
   public void escribe(String a, int n)
   {
     System.out.print("Método clase derivada, dos argumentos: ");
     System.out.println(a + "  " + n);
   }
}

public class PruebaSobre
{
   public static void main(String [] ar)
   {
     DerivSobre dr = new DerivSobre();
     dr.escribe("Cadena constante ",50);
     dr.escribe("Cadena constante ");
     dr.escribe(50);
   }
}
```

Ejecución

```
Método clase derivada, dos argumentos: Cadena constante  50
Método clase base, argumento cadena: Cadena constante
Método clase base, argumento entero: 50
```

4.2. HERENCIA PÚBLICA

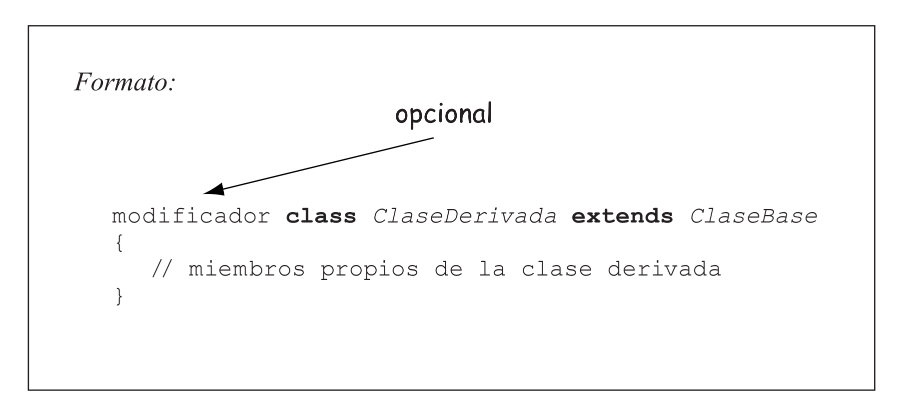
En una clase existen secciones *públicas*, *privadas*, *protegidas* y con la visibilidad por defecto, que se denomina *amigable*. Java considera que la herencia es siempre pública. *Herencia pública* significa que una clase derivada tiene acceso a los elementos públicos y protegidos de su clase base, mientras que los elementos con visibilidad *amigable* son accesibles desde cualquier clase del mismo paquete, pero no son visibles en clases derivadas de otros paquetes.

Una clase derivada no puede acceder a variables y métodos privados de su clase base. Una clase base utiliza elementos protegidos para, de esa manera, ocultar los detalles de la clase respecto a clases no derivadas de otros paquetes.

Tabla 4.1 Acceso a variables y métodos según *visibilidad*

Tipo de elemento	¿Accesible a clase de paquete (package)?	¿Accesible a clase derivada?	¿Accesible a clase derivada de otro paquete?
public	*sí*	*sí*	*sí*
protected	*sí*	*sí*	*sí*
private	*no*	*no*	*no*
(default)	*sí*	*sí*	*no*

Las clases para ser visibles desde otro paquete se declaran con el modificador **`public`**, en caso contrario, la clase está restringida al paquete donde se declara.



Ejercicio 4.2

Considérese la siguiente jerarquía:

ObjGeometrico

Cuadrado

Círculo

Las declaración de las clases se agrupan en el paquete `figuras`.

```
package figuras;

public class ObjGeometrico
{
   public  ObjGeometrico(double x, double y)
   {
```

```
      px = x;
      py = y;
   }
   public ObjGeometrico()
   {
      px = py = 0;
   }
   public void imprimirCentro()
   {
      System.out.println("(" + px + "," + py + ")");
   }
   protected double px, py;
}
```

Un círculo se caracteriza por el centro y por su radio. Un cuadrado, también por su centro y por uno de sus cuatro vértices. Entonces, las clases `Circulo` y `Cuadrado` se declaran derivadas de `ObGeometrico`.

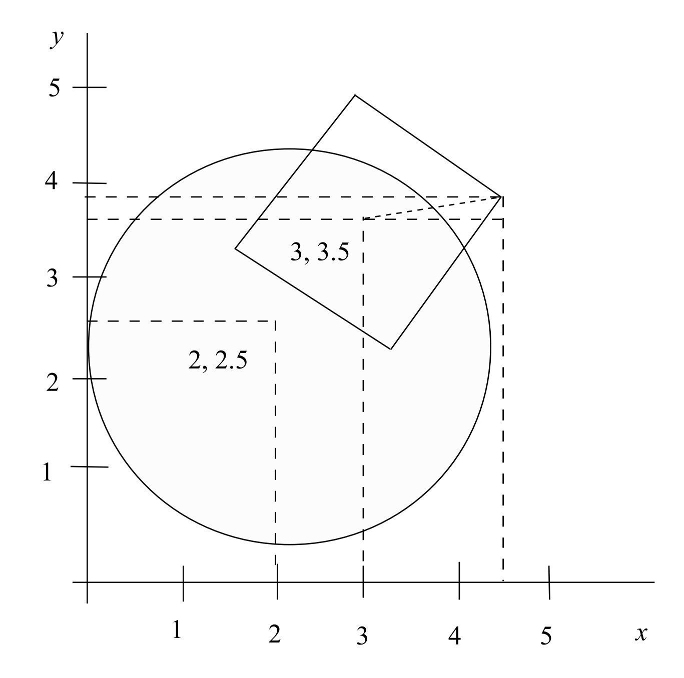


Figura 4.2 Círculo (centro: 2, 2.5), cuadrado (centro: 3, 3.5)

```
package figuras;

public class Circulo extends ObjGeometrico
{
   public  Circulo(double x, double y, double r)
   {
      super(x,y);    // llama a constructor de la clase base
      radio = r;
   }
```

```
    public double area()
    {
        return PI * radio * radio;
    }
    private double radio;
    private final double PI = 3.14159;
}

package figuras;

public class Cuadrado extends ObjGeometrico
{
    public  Cuadrado(double xc, double yc, double t1, double t2)
    {
        super(xc,yc); // llama a constructor de la clase base
        x1 = t1;
        y1 = t2;
    }
    public double area()
    {
        double a, b;
        a = px - x1;
        b = py - y1;
        return 2 * (a * a + b * b);
    }
    private double x1, y1;
}
```

Todos los miembros públicos de la clase base `ObjGeometrico` también son públicos de la clase derivada `Cuadrado`. Por ejemplo, se puede ejecutar:

```
Cuadrado c = new Cuadrado(3, 3.5, 4.37, 3.85);
c.imprimirCentro();
```

La siguiente aplicación utiliza las clases `Cuadrado` y `Circulo`:

```
import java.io.*;
import figuras.*;

public class PruebaFiguras
{
    public static void main(String[] ar)
    {
        Circulo cr = new Circulo(2.0, 2.5, 2.0);
        Cuadrado cd = new Cuadrado(3.0, 3.5, 4.37, 3.85);
        System.out.print("Centro del circulo  : ");
        cr.imprimirCentro();
        System.out.println("Centro del cuadrado : ");
        cd.imprimirCentro();
        System.out.println("Area del circulo : " + cr.area());
        System.out.println("Area del cuadrado : " + cd.area());
    }
}
```

Ejecución

```
Centro del circulo : (2.0,2.5)
Centro del cuadrado : (3.0,3.5)
Area del circulo : 12.5666
Area del cuadrado : 3.9988
```

> ***Regla***
>
> La herencia en Java es siempre pública. Los miembros de la clase derivada heredados de la clase base tienen la misma protección que en la clase base. La herencia pública modela directamente la relación ***es-un***.

4.3. CONSTRUCTORES EN HERENCIA

Un objeto de una clase derivada consta de la porción correspondiente de su clase base y de los miembros propios. En consecuencia, al construir un objeto de clase derivada, primero se construye la parte de su clase base, llamando a su constructor, y a continuación se inicializan los miembros propios de la clase derivada.

> ***Regla***
>
> 1. El constructor de la clase base se invoca antes del constructor de la clase derivada.
> 2. Si una clase base es, a su vez, una clase derivada, se invocan siguiendo la misma secuencia: constructor base, constructor derivada.
> 3. Los métodos que implementan a los constructores no se heredan.
> 4. Si no se especifica el constructor de la clase base, se invoca el constructor sin argumentos.

Ejemplo 4.4

Se declaran dos clases base, una clase derivada de una clase base y una clase derivada de una clase base que a su vez es derivada.

```
class B1
{
   public  B1() { System.out.println("Constructor-B1"); }
}
class B2
{
   public  B2() { System.out.println("Constructor-B2");}
}
class D extends B1
{
   public  D() { System.out.println("Constructor-D"); }
}
```

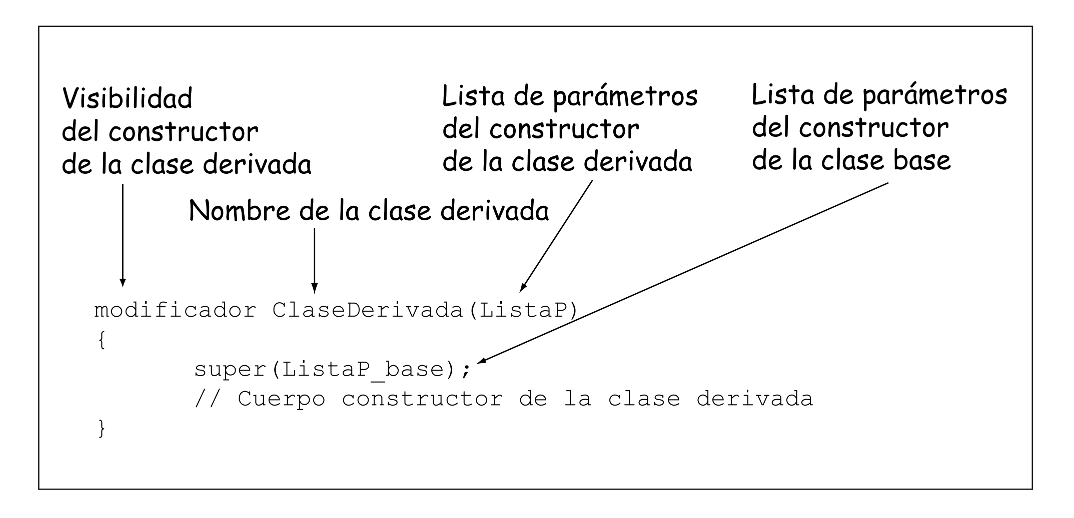
```
class H extends B2
{
   public  H() { System.out.println("Constructor-H"); }
}
class Dh extends H
{
   public  Dh() { System.out.println("Constructor-Dh"); }
}
class Constructor
{
   public static void main(String [] ar)
   {
      D d1 = new D();
      System.out.println("________________\n");
      Dh d2 = new Dh();
   }
}
```

Ejecución

```
Constructor-B1
Constructor-D
________________
Constructor-B2
Constructor-H
Constructor-Dh
```

4.3.1. Sintaxis

La primera línea del constructor de la clase derivada debe incluir una llamada al constructor de la clase base, que se hace a través de **`super`**`()`. Los argumentos que se vayan a transmitir se incluyen en la lista de argumentos de la clase base.



Por ejemplo:

```
class Persona
{
   protected  String nombre;
   public Persona (String nm) // constructor de Persona
   {
      nombre = new String(nm);
   }
}

class Juvenil extends Persona
{
   private int edad;
   public Juvenil (String suNombre, int ed)
   {
      super(suNombre);   // llamada a constructor de clase base
      edad = ed;
   }
}
```

La llamada al constructor de la clase base ha de ser la primera sentencia del cuerpo del constructor de la clase derivada. Si éste no tiene la llamada explícita, asume que se hace una llamada al constructor sin argumentos de la clase base, pudiendo dar error si no existiera tal constructor.

Ejemplo 4.5

La clase `Punto3D` *es una clase derivada de la clase* `Punto`.

En la clase `Punto3D` se definen dos constructores. El primero, sin argumentos, que inicializa un objeto al punto tridimensional `(0,0,0)`; esto lo hace en dos etapas, primero llama al constructor por defecto de `Punto` y, a continuación, asigna `0` a `z` (tercera coordenada). El segundo constructor llama, con `super(x1,y1)`, al constructor de `Punto`.

```
class Punto
{
   public  Punto()
   {
      x = y = 0;
   }
   public  Punto(int xv, int yv)
   {
      x = xv;
      y = yv;
   }
   protected int x, y;
}

class Punto3D extends Punto
{
   public  Punto3D()
   {
         // llamada implícita al constructor por defecto de Punto
         // Se podría llamar explícitamente: super(0,0);
      fijarZ(0);
```

```
    }
    public Punto3D(int x1, int y1, int z1)
    {
       super(x1,y1);
       fijarZ(z1);
    }
    private void fijarZ(int z) {this.z = z;}
    private int z;
}
```

4.3.2. Referencia a la clase base: `super`

Los métodos heredados de la clase base pueden ser llamados desde cualquier método de la clase derivada, simplemente se escribe su nombre y la lista de argumentos. Puede ocurrir que haya métodos de la clase base que no interese que sean heredados en la clase derivada, debido a que se quiera que tenga una funcionaldad adicional. Para ello, se sobreescribe el método en la clase derivada. Los métodos en las clases derivadas con la *misma signatura (igual nombre, tipo de retorno y número y tipo de argumentos)* que métodos de la clase base *anulan* (reemplazan) las versiones de la clase base. El método de la clase base ocultado puede ser llamado desde cualquier método de la clase derivada con la referencia **`super`** seguida de un punto (.) el nombre del método y la lista de argumentos: `super.metodo(argumentos)`. La palabra reservada **`super`** permite acceder a cualquier miembro de la clase base (siempre que no sea privado).

> ***Regla***
>
> Una clase derivada oculta un método de la clase base redefiniéndolo con la misma signatura: mismo nombre, igual tipo de retorno y la misma lista de argumentos.

Ejemplo 4.6

La clase `Fecha` *define el método* `escribir()`, *la clase* `FechaJuliana` *hereda de la clase* `Fecha` *y sobreescribe el método.*

```
class Fecha
{
   private int d, m, a;
   public Fecha(int dia, int mes, int anyo)
   {
      d = dia;
      m = mes ;
      a = anyo;
   }
   public Fecha(){}
   public void escribir()
   {
      System.out.println("\n" + d +" / " + m + " / " + a);
   }
}

class FechaJuliana extends Fecha
{
```

```
    private int numDias;
    public FechaJuliana(int dia, int mes, int anyo){...}
    public void escribir()
    {
       super.escribir();  // llamada al método de la clase Fecha
       System.out.println("Dias transcurridos: " + numDias);
    }
 }
```

4.4. MÉTODOS Y CLASES NO DERIVABLES: ATRIBUTO `final`

En el contexto de herencia, la palabra reservada `final` se emplea para proteger la redefinición de los métodos de la clase base. Un método con el atributo `final` no puede ser redefinido en las clases derivadas. Por ejemplo:

```
class Ventana
{
   final public int numpixels()
   {
      //...
   }
   public void rellenar()
   {
      //...
   }
}
```

El método `numpixeles()` no puede se redefinido por una clase derivada de `Ventana`, y si el compilador de Java detecta un intento de redefinición, genera un mensaje de error.

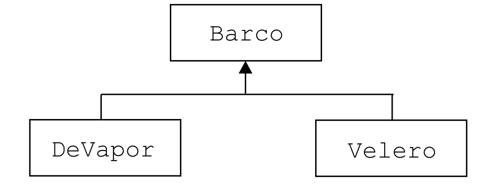
El concepto de protección que implica `final` también se extiende a las clases. Una clase que no se quiere que sea clase base de otras clases se declara con el atributo `final`.

```
final class Sorteo
{
   //...
}
```

La clase `Sorteo` no puede estar formando parte de una una jerarquía de clases, ha de ser una clase independiente. Las clases que *envuelven* los tipos básicos `(Integer, Boolean...)` son clases que no se pueden derivar, están declaradas como `final`.

4.5. CONVERSIONES ENTRE OBJETOS DE CLASE DERIVADA Y CLASE BASE

Al declarar una clase como *extensión* o *derivada* de otra clase, los objetos de la clase derivada son a su vez objetos de la clase base. En la siguiente jerarquía:



un objeto `Velero` puede verse según la Figura 4.3.

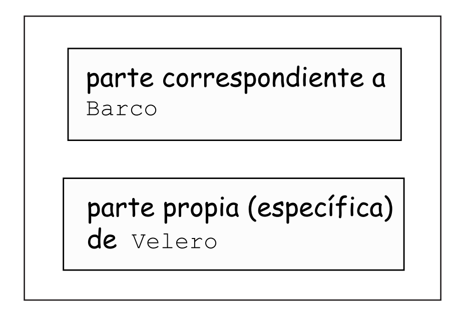


Figura 4.3 Ejemplo de objeto derivado

Un objeto `Velero` es a su vez un objeto `Barco`. Entonces, se puede afirmar que todo `Velero` es un `Barco`. Por esa razón, el lenguaje Java convierte automáticamente una referencia a objeto de la clase derivada a referecia a clase base. Por ejemplo:

```
Barco mr;
Velero v = new Velero();
DeVapor w = new DeVapor();
mr =  v; // conversión automática
mr =  w; // conversión automática
```

La conversión inversa, de un objeto de la clase base a referencia de la clase derivada, no es posible, es un error:

```
Barco mr = new Barco();
Velero v = new Velero();
v = mr; // es erronea esta conversión.
```

Ejercicio 4.3

Se declaran las clases correspondientes a la jerarquía `Barco`, `DeVapor` *y* `Velero`*; con un método común,* `alarma()`*, que es redefinido en cada clase derivada. En el programa se define un array de referencias a* `Barco`*, se crean objetos de las clases derivadas* `Devapor` *y* `Velero`*, asigna esos objetos al array y por último se llama al método redefinido.*

```
class Barco
{
   public Barco()
   {
      System.out.print("\tSe crea parte de un barco. ");
   }
   public void alarma()
   {
      System.out.println("\tS.O.S desde un Barco");
   }

}
class DeVapor extends Barco
```

```
{
   public DeVapor()
   {
      System.out.println("Se crea la parte del barco de vapor. ");
   }
   public void alarma()
   {
      System.out.println("\tS.O.S desde un Barco de Vapor");
   }
}

class Velero extends Barco
{
   public Velero()
   {
      System.out.println("Se crea la parte del barco velero. ");
   }
   public void alarma()
   {
      System.out.println("\tS.O.S desde un Velero");
   }
}

public class AlarmasDeBarcos
{
   public static void main(String [] ar)
   {
      Barco[] bs = new Barco[2];
      DeVapor mss = new DeVapor();
      Velero vss = new Velero();
      bs[0] = mss; bs[1] = vss;
      for (int i = 0; i<2; )  bs[i++].alarma();
   }
}
```

Ejecución:

```
Se crea parte de un barco. Se crea la parte del barco de vapor.
Se crea parte de un barco. Se crea la parte del barco velero.
S.O.S desde un Barco de Vapor
S.O.S desde un Velero
```

4.6. MÉTODOS ABSTRACTOS

Si la palabra reservada `abstract` precede a la declaración de un metodo, este método se denomina *abstracto*, y le indica al compilador que será definido (implementado su cuerpo) en una clase derivada (no necesariamente en la derivada inmediata). El uso común de los métodos abstractos es la declaración de clases abstractas y la implementación del polimorfismo.

Por ejemplo, en el contexto de figuras geométricas, la clase `Figura` es la clase base de la que derivan otras, como `Rectangulo`, `Circulo` y `Triangulo`. Cada figura debe tener la posibilidad de calcular su área y poder dibujarla; por ello, la clase `Figura` declara los métodos `calcularArea()` y `dibujar()` abstractos, y así obliga a las clases derivadas a redefinirlos.

```
class Figura
{
   public abstract double calcularArea();
   public abstract void dibujar();

   // otros métodos miembro que definen una interfaz a todos los
   // tipos de figuras geométricas
}
```

La clase `Figura` es una clase abstracta. Toda clase con uno o más métodos abstractos es abstracta y en Java se declara con la palabra reservada `abstract`. La declaración correcta de `Figura`:

```
abstract class Figura
```

Las clases `Circulo` y `Rectangulo`, derivadas de `Figura`, deben definir los métodos `calcularArea()` y `Dibujar()` en cada clase. Para la clase `Circulo`:

```
class Circulo extends Figura
{
   public  double calcularArea()
   {
      return (PI * radio * radio);
   }
   public void dibujar()
   {
      // ...
   }
   private double xc, yc; // coordenada del centro
   private double radio; // radio del círculo
}
```

Una clase que no redefina un método abstracto heredado se convierte en clase abstracta.

4.6.1. Clases abstractas

Las clases abstractas representan conceptos generales, engloban las características comunes de un conjunto de objetos. `Persona`, en un contexto de trabajadores, es una clase abstracta que engloba las propiedades y métodos comunes a todo tipo de persona que trabaja para una empresa. En Java, el modificador `abstract` declara una clase abstracta:

```
abstract class NombreClase { // ... }
```

Por ejemplo,

```
public abstract class  Persona
{
   private String apellido;
      //
   public void identificacion(String a, String c){ ... }
}
```

Las clases abstractas declaran métodos y variables instancia, y normalmente tienen métodos abstractos. Una clase que tiene un método abstracto debe declarse abstracta.

Una característica importante es que no se pueden definir objetos, *instanciar,* de una clase abstracta. El compilador da un error siempre que se intenta crear un objeto de una clase abstracta.

```
public abstract class Metal { ... }
Metal mer = new Metal(); // error: no se puede instanciar de clase
                         // abstracta
```

Las clases abstractas están en lo mas alto de la jerarquía de clases, son superclases base, y por consiguiente siempre se establece una conversión automática de clase derivada a clase base abstracta.

Ejemplo 4.7

Se define un array de la clase abstracta `Figura` *y se crean objetos de las clase concretas* `Rectangulo` *y* `Circulo`.

```
Figura []af = new Figura[10];
for (int i = 0; i < 10; i++)
{
   if (i % 2 ==0)
      af[i] = new Rectangulo();
   else
      af[i] = new Circulo();
}
```

Normas de las clases abstractas

- Una clase abstracta se declara con la palabra reservada `abstract` como prefijo en la cabecera de la clase.
- Una clase con al menos un método abstracto es una clase abstracta y hay que declararla como tal.
- Una clase derivada que no redefine un método abstracto es también una clase abstracta.
- Las clases abstractas pueden tener variables instancia y métodos no abstractos.
- No se pueden crear objetos de clases abstractas.

4.7. POLIMORFISMO

En POO, el *polimorfismo* permite que diferentes objetos respondan de modo diferente al mismo mensaje. El polimorfismo adquiere su máxima potencia cuando se utiliza en unión de herencia.

El polimorfismo se establece con la ligadura dinámica de métodos. Con la ligadura dinámica, no es preciso decidir el tipo de objeto hasta el momento de la ejecución. En el Ejemplo 4.4 se declara `abstract` el método `dinamica()` de la clase `A`; se ha indicado al compilador que este método se puede llamar por una referencia de `A` mediante la ligadura dinámica. La variable `a` en un momento referencia a un objeto de `B` y en otro momento a un objeto de `C`. El programa determina

el tipo de objeto de `a` en tiempo de ejecución, de tal forma que el mismo *mensaje,* `dinamica()`, se comporta de manera diferente según `a` referencie a un objeto de `B` o a un objeto de `C`.

El polimorfismo se puede representar con un array de elementos que se refieren a objetos de diferentes tipos (clases), como sugiere Meyer[1].

4.7.1. Uso del polimorfismo

La forma de usar el polimorfismo es a través de referencias a la clase base. Si, por ejemplo, se dispone de una colección de objetos `Archivo` en un array, éste almacena referencias a objetos `Archivo` que apuntan a cualquier tipo de archivo. Cuando se actúa sobre estos archivos, simplemente basta con recorrer el array e invocar el método apropiado mediante la referencia a la instancia. Naturalmente, para realizar esta tarea los métodos de la clase base deben ser declarados como abstractos en la clase `Archivo`, que es la clase base, y redefinirse en las clases derivadas `(ASCII, Grafico...)`.

Para poder utilizar polimorfismo en Java se deben seguir las siguientes reglas:

1. Crear una jerarquía de clases con las operaciones importantes definidas por los métodos miembro declarados como abstractos en la clase base.
2. Las implementaciones específicas de los métodos abstractos se deben hacer en las clases derivadas. Cada clase derivada puede tener su propia versión del método. Por ejemplo, la implementación del método `annadir()` varía de un tipo de archivo a otro.
3. Las instancias de estas clases se manejan a través de una referencia a la clase base. Este mecanismo es la ligadura dinámica, que es la esencia del polimorfismo en Java.

Realmente, no es necesario declarar los métodos en la clase base como abstractos, si después se redefinen (misma *signatura*) en la clase derivada.

4.7.2. Ventajas del polimorfismo

El polimorfismo hace su sistema más flexible sin perder ninguna de las ventajas de la compilación estática de tipos que tienen lugar en tiempo de compilación. Este es el caso de Java.

Las aplicaciones más frecuentes del polimorfismo son:

- **Especialización de clases derivadas**. El uso más común del polimorfismo es derivar clases especializadas de clases que han sido definidas. Así, por ejemplo, una clase `Cuadrado` es una especialización de la clase `Rectangulo` (cualquier cuadrado es un tipo de rectángulo). Esta clase de polimorfismo aumenta la eficiencia de la subclase, mientras conserva un alto grado de flexibilidad y permite un medio uniforme de manejar rectángulos y cuadrados.
- **Estructuras de datos heterogéneos**. A veces es muy útil poder manipular conjuntos similares de objetos. Con polimorfismo se pueden crear y manejar fácilmente estructuras de datos heterogéneos, que son fáciles de diseñar y dibujar, sin perder la comprobación de tipos de los elementos utilizados.
- **Gestión de una jerarquía de clases**. Las jerarquías de clases son colecciones de clases altamente estructuradas, con relaciones de herencia que se pueden extender fácilmente.

---

[1] Meyer, B.: *Object-Oriented Software Construction*. Prentice-Hall, New York, 1998 (libro traducido al español por un equipo de profesores dirigido por el profesor Luis Joyanes).

4.8. INTERFACES

Java incorpora una construcción del lenguaje, llamada `interface`, que permite declarar un conjunto de constantes y de cabeceras de métodos abstractos. Estos deben implementarse en las clases y constituyen su interfaz. En cierto modo, es una forma de declarar que todos los métodos de una clase son públicos y abstractos, con lo que se especifica el comportamiento común de todas las clases que implementen la interfaz.

La declaración de una *interfaz* es similar a la de una clase; en la cabecera se utiliza la palabra reservada `interface` en vez de `class`, por ejemplo:

```
public interface NodoG
{
   boolean igual(NodoG t);
   NodoG asignar(NodoG t);
   void escribir(NodoG t);
}
```

La interfaz `NodoG` define tres métodos abstractos y además públicos. Sin embargo no es necesario especificar ni `abstract` ni `public` ya que todos los métodos de una `interface` lo son.

Sintaxis

```
acceso interface NombreInterface
{
   constante_1;
   ...
   constante_n;

   tipo_1 nombreMetodo_1(argumentos);
   ...
   tipo_n nombreMetodo_n (argumentos);
}
```

acceso es la visibilidad de la interfaz definida, normalmente `public`.

> ***Regla***
>
> En una `interface`, todos los métodos declarados son, por defecto, públicos y abstractos; por ello no es necesario especificar `public` ni `abstract`.

Ejemplo 4.8

Se declara un conjunto de métodos comunes a la estructura ArbolB; además, la constante entera que indica el número máximo de claves.

```
public interface  ArbolBAbstracto
{
   final int MAXCLAVES = 4;
   void insertar(Object clave);
   void eliminar(Object clave);
   void recorrer();
}
```

Esta interfaz muestra los métodos que todo árbol B debe implementar.

4.8.1. Implementación de una interfaz

La interfaz especifica el comportamiento común que tiene un conjunto de clases. Dicho comportamiento se implementa en cada una de las clases, es lo que se entiende como *implementación de una interfaz*. Se utiliza una sintaxis similar a la derivación o extensión de una clase, con la palabra reservada `implements` en lugar de `extends`.

```
class NombreClase implements NombreInterfaz
{
   // definición de atributos
   // implementación de métodos de la clase
   //    implementación de métodos de la interfaz
}
```

La clase que implementa una interfaz tiene que especificar el código (la implementación) de cada uno de los métodos de la interfaz. De no hacerlo la clase se convierte en clase abstracta y entonces debe declararse `abstract`. Es una forma de obligar a que cada método de la interfaz se implemente.

Ejercicio 4.4

Considérese una jerarquía de barcos, todos tienen como comportamiento común `msgeSocorro()` y `alarma()`. Las clases `BarcoPasaje`, `PortaAvion` y `Pesquero` implementan el comportamiento común.

Se declara la interfaz `Barco`:

```
interface Barco
{
   void alarma();
   void msgeSocorro(String av);
}
```

Las clases `BarcoPasaje`, `PortaAvion` y `Pesquero` implementan la interfaz Barco y, además, sus métodos:

```
class BarcoPasaje implements Barco
{
   private int eslora;
   private int numeroCamas = 101;
   public BarcoPasaje()
   {
      System.out.println("Se crea objeto BarcoPasaje.");
   }
   public void alarma()
   {
      System.out.println("¡¡¡ Alarma del barco pasajero !!!");
   }
   public void msgeSocorro(String av)
   {
      alarma();
      System.out.println("¡¡¡ SOS SOS !!!" + av);
   }
}
```

```
class PortaAvion implements Barco
{
   private int aviones = 19;
   private int tripulacion;
   public PortaAvion(int marinos)
   {
      tripulacion = marinos;
      System.out.println("Se crea objeto PortaAviones.");
   }
   public void alarma()
   {
      System.out.println("¡¡¡ marineros a sus puestos !!!");
   }
   public void msgeSocorro(String av)
   {
      System.out.println("¡¡¡ SOS SOS !!! " + av);
   }
}
class Pesquero implements Barco
{
   private int eslora;
   private double potencia;
   private int pescadores;
   String nombre;
   public Pesquero(int tripulacion)
   {
      pescadores = tripulacion;
      System.out.println("Se crea objeto Barco Pesquero.");
   }
   public void alarma()
   {
      System.out.println("¡¡¡ Alarma desde el pesquero  " +
                              nombre + " !!!");
   }
   public void msgeSocorro(String av)
   {
      System.out.println("¡¡¡ SOS SOS !!! " + av);
   }
}
```

Múltiples interfaces

Java no permite que una clase derive de dos o más clases, no permite la herencia múltiple. Sin embargo, una clase sí puede implementar más de una interfaz, sí puede tener el comportamiento común de varias interfaces. Sencillamente, a continuación de la palabra reservada `implements` se escriben las interfaces separadas por comas. La clase tiene que implementar los métodos de todas las interfaces.

Sintaxis

```
class NombreClase implements Interfaz1, Interfaz2,…,Interfazn
{
   // …
}
```

> ***Regla***
>
> Una clase implementa tantas interfaces como se desee. Se deben implementar todos los métodos como `public` debido a que Java no permite reducir la visibilidad de un método cuando se *sobreescibe*.

4.8.2. Jerarquía de interfaz

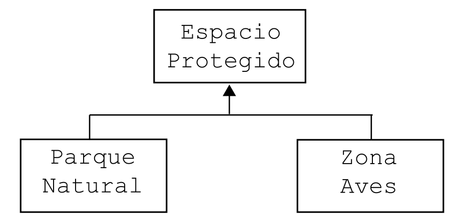
Las interfaces se pueden organizar en forma jerárquica, de tal forma que los métodos sean heredados. A diferencia de las clases que sólo pueden heredar de una clase base (*herencia simple*), las interfaces pueden heredar de tantos interfaces como se precise. También se utiliza la palabra reservada `extends` para especificar la herencia de la interfaz.

Sintaxis

```
interface SuperBase1 {…}
interface Base1 extends SuperBase1 {…}
interface Base2 extends SuperBase1 {…}
interface ComunDerivado extends Base1, Base2 {…}
```

4.8.3. Herencia de clases e implementación de interfaz

Las interfaces no son clases, especifican un comportamiento (métodos) que va a tener la clase que lo implementa. Por ello, una clase puede heredar de su clase base y a la vez implementar una interfaz. En la siguiente jerarquía de clases:



la clase `ParqueNatural` hereda de la clase `EspacioProtegido` y además implementa la interfaz `Parque`. El esquema para implementar este diseño es el siguiente:

```
public interface Parque {…}
public class EspacioProtegido {…}
public class ZonaAves extends EspacioProtegido {…}
public class ParqueNatural extends EspacioProtegido implements Parque{…}
```

> ***Regla***
>
> Una clase puede heredar de otra clase e implementar un interfaz. Se ha de especificar en primer lugar la clase de la que hereda `(extends)` y a continuación la interfaz que implementa `(implements)`.

4.8.4. Variables interfaz

Las `interfaces`, al no ser clases, tampoco pueden instanciar objetos. Sí se pueden declarar variables de tipo `interface`; cualquier variable de una clase que implementa a una `interfaz` se puede asignar a una variable del tipo de la `interfaz`.

Ejemplo 4.9

Se declaran la interfaz Bolsa y las clases que implementan la interfaz.

```
interface Bolsa
{
   Bolsa insertar (Object elemento);
}

public class Bolsa1 implements Bolsa
{
   public Bolsa insertar(Object e) { ... }
}

public class Bolsa2 implements Bolsa
{
   public Bolsa insertar(Object e) { ... }
}
```

En un método se puede definir una variable del tipo `interface Bolsa` y asignar un objeto `Bolsa1`, o bien un objeto `Bolsa2`

```
Bolsa q;
q = new Bolsa1();
q.insertar("Manzana");
   ...
q = new Bolsa2();
q.insertar(new Integer(5));
```

RESUMEN

La relación entre clases **`es-un tipo de`** indica relación de herencia. Por ejemplo, **`una`** revista **`es un tipo de`** publicación. La relación **`es-un`** también se puede expresar como generalización-especialización, es una relación transitiva; así, un *becario* es un tipo de *trabajador* y un *trabajador,* un tipo de *persona*; por consiguiente, un *becario* es una *persona*. Esta manera de relacionarse las clases se expresa en Java con la derivación o extensión de clases.

Una clase nueva que se crea a partir de una clase ya existente, utilizando la propiedad de la herencia, se denomina *clase derivada* o *subclase*. La clase de la cual se hereda se denomina *clase base* o *superclase*.

La herencia puede ser simple o múltiple. La herencia simple entre clases se produce cuando una nueva clase se define utilizando las propiedades de una clase ya definida; la nueva clase es la clase derivada. En Java, la herencia siempre es simple y, además, siempre es pública. La clase derivada hereda todos los miembros de la clase base excepto los **miembro privados.**

La relación de herencia reduce código redundante en los programas o aplicaciones. En el paradigma de la orientación a objetos, para que un lenguaje sea considerado orientado a objetos, debe soportar la propiedad de la herencia.

La herencia múltiple se produce cuando una clase deriva de dos o más clases base. Este tipo de herencia crea problemas por las colisiones o conflictos de nombres, esa es una de las razones por las que Java no incorpora la herencia múltiple. En caso de considerarse necesario se puede simular herencia múltiple con los interfaces.

Un objeto de una clase derivada se crea siguiendo este orden: primero la parte del objeto correspondiente a la clase base y, a continuación, se crea la parte propia de la clase derivada. Para llamar al constructor de la clase base desde el constructor de la clase derivada se emplea la palabrea reservada **`super()`**.

El polimorfismo es una de las propiedades fundamentales de la orientación a objetos. Esta propiedad significa que el envío de un *mensaje* puede dar lugar a acciones diferentes dependiendo del objeto que lo reciba.

Para implementar el polimorfismo, un lenguaje debe soportar el enlace entre la llamada a un método y el código del método en tiempo de ejecución; es la **ligadura dinámica** o **vinculación tardía.** Esta propiedad se establece en el contexto de la herencia y de la redefinición de los métodos polimórficos en cada clase derivada.

Un método abstracto (**`abstract`**) declarado en una clase convierte esta en una clase abstracta. Con los métodos abstractos se obliga a su redifinición en la clase derivada, en caso contrario, la clase derivada también es abstracta. No se puede instanciar objetos de clases abstractas.

Java pemite declarar métodos con la propiedad de no ser redefinibles, por medio del modificador **`final`**. También, con el modificador **`final`** se puede hacer que una clase no forme parte de una jerarquía.

Las *interfaces* de Java declaran constantes y operaciones comunes a un conjunto de clases. Las operaciones son métodos abstractos que deben definir las clases que implementan el `interface`.

EJERCICIOS

4.1. Implementar una clase `Automovil (Carro)` dentro de una jerarquía de herencia. Considere que, además de ser un *Vehículo,* un automóvil es también una *comodidad, un símbolo de estado social, un modo de transporte,* etc.

4.2. Implementar una jerarquía de herencia de animales que contenga al menos seis niveles de derivación y doce clases.

4.3. Deducir las clases necesarias para diseñar un programa de ordenador que permita jugar a diferentes juegos de cartas.

4.4. Implementar una jerarquía de clases de los distintos tipos de ficheros. Codificar en Java la cabecera de las clases y los métodos que se consideren polimórficos.

4.5. ¿Describir las diversas utulizaciones de la referencia `super`?

4.6. ¿Qué diferencias se pueden encontrar entre `this` y `super`?

4.7. Declarar una interfaz con el comportamiento común de los objetos *Avión*.

4.8. Declarar la interfaz `Conjunto` con las operaciones que puede realizar todo tipo de conjunto.

PROBLEMAS

4.1. Definir una clase base `Persona` que contenga información de propósito general común a todas las personas (nombre, dirección, fecha de nacimiento, sexo, etc.). Diseñar una jerarquía de clases que contemple las clases siguientes: `Estudiante`, `Empleado`, `Estudiante _ Empleado`.

Escribir un programa que lea del dispositivo estándar de entrada los datos para crear una lista de personas: a) general; b) estudiantes; c) empleados; d) estudiantes empleados. El programa deber permitir ordenar alfabéticamente por el primer apellido.

4.2. Implementar una jerarquía `Librería` que tenga al menos una docena de clases. Considérese una *librería* con colecciones de libros de literatura, humanidades, tecnología, etc.

4.3. Diseñar una jerarquía de clases que utilice como clase base o raíz una clase LAN (red de área local).

Las subclases derivadas deben representar diferentes topologías, como *estrella, anillo, bus y hub*. Los miembros datos deben representar propiedades tales como *soporte de transmisión, control de acceso, formato del marco de datos, estándares, velocidad de transmisión,* etc. Se desea simular la actividad de los nodos de la LAN.

La red consta de **nodos,** que pueden ser dispositivos tales como computadoras personales, estaciones de trabajo, máquinas fax, etc. Una tarea principal de LAN es soportar comunicaciones de datos entre sus nodos. El usuario del proceso de simulación debe, como mínimo, poder:

- Enumerar los nodos actuales de la red LAN.
- Añadir un nuevo nodo a la red LAN.
- Quitar un nodo de la red LAN.
- Configurar la red, proporcionándole una topología de *estrella* o en *bus*.
- Especificar el tamaño del paquete, que es el tamaño en bytes del mensaje que va de un nodo a otro.
- Enviar un paquete de un nodo especificado a otro.
- Difundir un paquete desde un nodo a todos los demás de la red.
- Realizar estadísticas de la LAN, como tiempo medio que emplea un paquete.

4.4. Implementar una jerarquía `Empleado` de cualquier tipo de empresa que le sea familiar. La jerarquía debe tener al menos tres niveles, con herencia de miembros dato y métodos. Los métodos deben poder calcular salarios, despidos, promociones, altas, jubilaciones, etc. Los métodos deben permitir también calcular aumentos salariales y primas para empleados de acuerdo con su categoría y productividad. La jerarquía de herencia debe poder ser utilizada para proporcionar diferentes tipos de acceso a empleados. Por ejemplo, el tipo de acceso garantizado al público diferirá del tipo de aceso proporcionado a un supervisor de empleado, al departamento de nóminas o al Ministerio de Hacienda.

4.5. Se quiere realizar una aplicación para que cada profesor de la universidad gestione las fichas de sus alumnos. Un profesor puede impartir una o varias asignaturas, y dentro de cada asignatura puede tener distintos grupos de alumnos. Los alumnos pueden ser *presenciales* o *a distancia*. Al comenzar las clases, se entrega al profesor un listado con los alumnos por cada asignatura.

Escribir un programa de tal forma que el listado de alumnos se introduzca por teclado y se den de alta calificaciones de examenes o prácticas realizadas. Se podrá obtener listados de calificaciones y porcentajes de aprobados una vez realizados los examenes.

4.6. Escribir una interfaz `FigGeometrica` que represente figuras geométricas como *punto, línea, rectángulo, triángulo* y similares. Debe proporcionar métodos que permitan dibujar, ampliar, mover y destruir tales objetos.

4.7. Implementar una jerarquía de tipos datos numéricos que extienda los tipos de datos fundamentales, tales como `int` y `float`, disponibles en Java. Las clases a diseñar pueden ser `Complejo`, `Racional`, etc.

4.8. Diseñar la siguiente jerarquía de clases:

	Persona		
	`nombre`		
	`edad`		
	`visualizar()`		
Estudiante		*Profesor*	
`nombre`	heredado	`nombre`	heredado
`edad`	heredado	`edad`	heredado
`id`	definido	`salario`	definido
`visualizar()`	*redefinido*	`visualizar`	*heredada*

Escribir un programa que manipule la jerarquía de clases, lea un objeto de cada clase y lo visualice:

a) Sin utilizar métodos abstractos.

b) Utilizando métodos abstractos.

4.9. El programa siguiente muestra las diferencias entre llamadas a un método redefinido y otro no.

```
class Base
{
   public void f(){System.out.println("f(): clase base !");}
   public void g(){System.out.println("g(): clase base !");}
}

class Derivada1 extends Base
{
   public void f()
      {System.out.println("f():clase Derivada !");}
   public void g(int k)
     {System.out.println( "g() :clase Derivada !" + k);}
}

class Derivada2 extends Derivada1
{
   public void f()
      {System.out.println( "f() :clase Derivada2 !";)}
   public  void g()
      {System.out.println( "g() :clase Derivada2 !" ;) }
};

class Anula
{
   public static void main(String ar[])
   {
      Base b = new Base();
      Derivada1 d1 = new Derivada1();
      Derivada2 d2 = new Derivada2();
      Base p = b;
      p.f();
      p.g();
      p = d1;
      p.f();
      p.g();
      p = d2;
      p.f();
      p.g();
   }
}
```

¿Cuál es el resultado de ejecutar este programa? ¿Por qué?

CAPITULO 5

Algoritmos recursivos

Objetivos

Una vez que haya leído y estudiado este capítulo, usted podrá:

- Conocer cómo funciona la recursividad.
- Distinguir entre recursividad y recursividad indirecta.
- Resolver problemas numéricos sencillos aplicando métodos recursivos básicos.
- Aplicar la técnica algorítmica *divide y vence* para la resolución de problemas.
- Determinar la complejidad de algoritmos recursivos mediante inducción matemática.
- Conocer la técnica algorítmica de resolución de problemas *vuelta atrás: backtracking*.
- Aplicar la técnica de *backtracking* al problema de la *selección óptima.*

Contenido

Conceptos clave

- *Backtracking*.
- Búsqueda exhaustiva.
- Caso base.
- Complejidad.
- *Divide y vence*.
- Inducción.
- Iteración versus recursión.
- Mejor selección.
- Recursividad.
- Torres de Hanoi.

Para profundizar (página web: www.mhe.es/joyanes)

- Ordenación por mezclas: *mergesort*.
- Resolución de problemas con algoritmos de vuelta atrás.

INTRODUCCIÓN

La *recursividad (recursión)* es aquella propiedad que posee un método por la cual puede llamarse a sí mismo. Aunque se puede utilizar la recursividad como una alternativa a la iteración, una solución recursiva es, normalmente, menos eficiente en términos de tiempo de computadora que una solución iterativa, debido a las operaciones auxiliares que llevan consigo las invocaciones suplementarias a los métodos; sin embargo, en muchas circunstancias, el uso de la recursión permite a los programadores especificar soluciones naturales, sencillas, que serían, en caso contrario, difíciles de resolver. Por esta causa, la recursión es una herramienta poderosa e importante en la resolución de problemas y en la programación. Diversas técnicas algorítmicas utilizan la recursión, como los algoritmos *divide y vence* y los algoritmos de *vuelta atrás*.

5.1. LA NATURALEZA DE LA RECURSIVIDAD

Los programas examinados hasta ahora, generalmente estructurados, se componen de una serie de métodos que se llaman de modo disciplinado. En algunos problemas es útil disponer de métodos que se llamen a sí mismos. Un método *recursivo* es aquel que se llama a sí mismo, bien directamente o bien indirectamente, a través de otro método. La recursividad es un tópico importante examinado frecuentemente en cursos que estudian la resolución de algoritmos y en cursos relativos a estructuras de datos.

En este libro se dará una importancia especial a las ideas conceptuales que soportan la recursividad. En matemáticas existen numerosas funciones que tienen carácter recursivo; de igual modo, numerosas circunstancias y situaciones de la vida ordinaria tienen carácter recursivo. Piense, por ejemplo, en la búsqueda de "Sierra de Lupiana" en páginas web, puede ocurrir que aparezcan direcciones (enlaces) que lleven a otras páginas y éstas, a su vez, a otras nuevas y así hasta completar todo lo relativo a la búsqueda inicial.

Un método que tiene sentencias entre las que se encuentra al menos una que llama al propio método se dice que es *recursivo.* Así, supongamos que se dispone de dos métodos `metodo1` y `metodos2`. La organización de una aplicación no recursiva adoptaría una forma similar a ésta:

```
metodo1 (...)
{
    ...
}

metodo2 (...)
{
    ...
    metodo1(); //llamada al metodo1
    ...
}
```

Con una organización recursiva, se tendría esta situación:

```
metodo1(...)
{
    ...
    metodo1();
    ...
}
```

Ejemplo 5.1

Algoritmo recursivo de la función matemática que suma los n primero números enteros positivos.

Como punto de partida, se puede afirmar que para `n = 1` se tiene que la suma `S(1) = 1.` Para `n = 2` se puede escribir `S(2) = S(1) + 2;` en general, y aplicando la inducción matemática, se tiene:

```
S(n) = S(n-1) + n
```

Se está definiendo la función suma `S()` respecto de sí misma, eso sí, siempre para un caso más pequeño. `S(2)` respecto a `S(1)`, `S(3)` respecto a `S(2)` y, en general `S(n)` respecto a `S(n-1)`.

El algoritmo que determina la suma de modo *recursivo* ha de tener presente una condición de salida o una condición de parada. Así, en el caso del cálculo de `S(6)`; la definición es `S(6) = = 6 + S(5)`, a su vez `S(5)` es `5 + S(4)`, este proceso continúa hasta `S(1) = 1` por definición. En matemáticas la definición de una función en términos de sí misma se denomina definición **inductiva** y, de forma natural, conduce a una implementación recursiva. El **caso base** `S(1) = 1` es esencial dado que se detiene, potencialmente, una cadena de llamadas recursivas. Este caso base o condición de salida debe fijarse en cada solución recursiva. La implementación del algoritmo es:

```
long sumaNenteros (int n)
{
   if (n == 1)
      return 1;
   else
      return n + sumaNenteros(n - 1);
}
```

Ejemplo 5.2

Definir la naturaleza recursiva de la ***serie de Fibonacci:*** *0, 1, 1, 2, 3, 5, 8 ,13, 21 ...*

Se observa en esta serie que comienza con 0 y 1, y tiene la propiedad de que cada elemento es la suma de los dos elementos anteriores, por ejemplo:

```
0 + 1 = 1
1 + 1 = 2
2 + 1 = 3
3 + 2 = 5
5 + 3 = 8
...
```

Entonces se puede establecer que :

```
fibonacci(0) = 0
fibonacci(1) = 1
...
fibonacci(n) = fibonacci(n - 1) + fibonacci(n - 2)
```

y la definición recursiva será :

```
fibonacci(n) = n                                   si n = 0 o n = 1
fibonacci(n) = fibonacci(n - 1) + fibonacci(n - 2) si n > = 2
```

Obsérvese que la definición recursiva de los números de Fibonacci es diferente de las definiciones recursivas del factorial de un número y del producto de dos números. Así:

```
fibonacci(6) = fibonacci(5) + fibonacci(4)
```

o, lo que es igual, `fibonacci(6)` ha de aplicarse en modo recursivo dos veces, y así sucesivamente.

El algoritmo iterativo equivalente es:

```
if (n == 0 || n == 1)
   return n;
fibinf = 0;
fibsup = 1;
for (i = 2; i <= n; i++)
{
   int x;
   x = fibinf;
   fibinf = fibsup;
   fibsup = x + fibinf;
}
return (fibsup);
```

El tiempo de ejecución del algoritmo crece linealmente con `n` ya que el bucle es el término dominante. Se puede afirmar que `t(n)` es `O(n)`.

El algoritmo recursivo es:

```
long fibonacci (long n)
{
   if (n == 0 || n == 1)
      return n;
   else
      return fibonacci(n - 1) + fibonacci(n - 2);
}
```

En cuanto al tiempo de ejecución del algoritmo recursivo, ya no es tan elemental establecer una cota superior. Observe que, por ejemplo, para calcular `fibonacci(6)` se calcula, recursívamente, `fibonacci(5)` y, cuando termine éste, `fibonacci(4)`. A su vez, el cálculo de `fibonacci(5)` supone calcular `fibonacci(4)` y `fibonacci(3)`; se está repitiendo el cálculo de `fibonacci(4)`, es una pérdida de tiempo. Por inducción matemática se puede demostrar que el número de llamadas recursivas crece exponencialmente, $t(n)$ es $O(2^n)$.

A tener en cuenta

La formulación recursiva de una función matemática puede ser muy ineficiente sobre todo si se repiten cálculos realizados anteriormente. En estos casos el algoritmo iterativo, aunque no sea tan evidente, es notablemente más eficiente.

5.2. MÉTODOS RECURSIVOS

Un método **recursivo** es un método que se invoca a sí mismo de forma directa o indirecta. En **recursión directa,** el código del método `f()` contiene una sentencia que invoca a `f()`, mientras

que en **recursión indirecta,** el método `f()` invoca a un método `g()` que a su vez invoca al método `p()`, y así sucesivamente hasta que se invoca de nuevo al método `f()`.

Un requisito para que un algoritmo recursivo sea correcto es que no genere una secuencia infinita de llamadas sobre sí mismo. Cualquier algoritmo que genere una secuencia de este tipo no puede terminar nunca. En consecuencia la definición recursiva debe incluir una *condición de salida*, que se denomina **componente base,** en el que *f(n)* se defina directamente (es decir, no recursivamente) para uno o más valores de *n*.

En definitiva, debe existir una *"forma de salir"* de la secuencia de llamadas recursivas. Así, en el algoritmo que calcula la suma de los n primeros enteros:

$$S(n) = \begin{cases} 1 & n = 1 \\ n + S(n-1) & n > 1 \end{cases}$$

la condición de salida o componente base es *S (n) = 1* para *n = 1.*

En el caso del algoritmo recursivo de la serie de Fibonacci:

$$F_0 = 0, \quad F_1 = 1; \; F_n = F_{n-1} + F_{n-2} \qquad \textit{para } n > 1$$

$F_0 = 0$ y $F_1 = 1$ constituyen el componente base o condiciones de salida, y $F_n = F_{n-1} + F_{n-2}$ es el componente recursivo. Un método recursivo correcto debe incluir un componente base o condición de salida ya que, en caso contrario, se produce una recursión infinita.

> ***A tener en cuenta***
>
> Un método es recursivo si se llama a sí mismo, directamente o bien indirectamente a través de otro método `g()`. Es necesario contemplar un caso base que determina la salida de las llamadas recursivas.

Ejercicio 5.1

Escribir un método recursivo que calcule el factorial de un número n y un programa que pida un número entero y escriba su factorial..

La *componente base* del método recursivo que calcula el factorial es que n = 0 o incluso n = 1, ya que en ambos caso el factorial es 1. El problema se resuelve recordando la definición expuesta anteriormente del factorial:

```
n! = 1              si n = 0   o n = 1    (componente base)
n! = n(n - 1)       si n > 1
```

En la implementación no se realiza tratamiento de error, que puede darse en el caso de calcular el factorial de un número negativo.

```
import java.io.*;

public class Factorial
{
```

```
    public static void main(String[] ar)throws IOException
    {
       int n;
       BufferedReader entrada = new BufferedReader(
                           new InputStreamReader(System.in));
       do {
          System.out.print("Introduzca número n: ");
          n = Integer.parseInt(entrada.readLine());
       }while (n < 0);
       System.out.println("\n \t" + n + "!= " + factorial(n));
    }
    static long factorial (int n)
    {
       if (n <= 1)
          return 1;
       else
       {
          long resultado = n * factorial(n - 1);
          return resultado;
       }
    }
 }
```

5.2.1. Recursividad indirecta: métodos mutuamente recursivos

La recursividad indirecta se produce cuando un método llama a otro, que eventualmente terminará llamando de nuevo al primer método.

Ejercicio 5.2

Mostrar por pantalla el alfabeto, utilizando recursión indirecta.

El método `main()` llama a **`metodoA()`** con el argumento '`Z`' (la última letra del alfabeto). Este examina su parámetro *c,* si *c* está en orden alfabético después que '`A`', llama a **`metodoB()`,** que inmediatamente invoca a **`metodoA()`** pasándole un parámetro predecesor de *c.* Esta acción hace que **`metodoA()`** vuelva a examinar c, y nuevamente llame a **`metodoB()`**. Las llamadas continúan hasta que *c* sea igual a '`A`'. En este momento, la recursión termina ejecutando **`System.out.print()`** veintiséis veces y visualiza el alfabeto, carácter a carácter.

```
public class Alfabeto
{
   public static void main(String [] a)
   {
      System.out.println();
      metodoA('Z');
      System.out.println();
   }
   static void metodoA(char c)
   {
      if (c > 'A')
```

```
            metodoB(c);
            System.out.print(c);
        }
        static void metodoB(char c)
        {
            metodoA(--c);
        }
    }
```

5.2.2. Condición de terminación de la recursión

Cuando se implementa un método recursivo, es preciso considerar una condición de terminación ya que, en caso contrario, continuaría indefinidamente llamándose a sí mismo y llegaría un momento en que la pila que registra las llamadas se desbordaría. En consecuencia, en cualquier método recursivo se necesita establecer la *condición de parada* de las llamadas recursivas y evitar indefinidas llamadas. Por ejemplo, en el caso del método `factorial()` definido anteriormente, la condición de parada ocurre cuando `n` es `1` o `0`, ya que en ambos casos el factorial es `1`. Es importante que cada llamada suponga un acercamiento a la condición de parada, porque en el método factorial cada llamada supone un decrecimiento del entero `n` lo que supone estar más cerca de la condición `n == 1`.

En el Ejercicio 5.2 se muestra una recursión mutua entre `metodoA()` y `metodoB()`, la condición de parada es que `c == 'A'`, y cada llamada mutua supone un acercamiento a la letra `'A'`.

> ***A tener en cuenta***
>
> En un algoritmo recursivo, se entiende por caso base el que se resuelve sin recursión, directamente con unas pocas sentencias elementales. El caso base se ejecuta cuando se alcanza la condición de parada de llamadas recursivas. Para que funcione la recursión el progreso de las llamadas debe tender a la condición de parada.

5.3. RECURSIÓN *VERSUS* ITERACIÓN

Se han estudiado varios métodos que se pueden implementar fácilmente, bien de modo recursivo, bien de modo iterativo. En esta sección se comparan los dos enfoques y se examinan las razones por las que el programador puede elegir un enfoque u otro según la situación específica.

Tanto la iteración como la recursión se basan en una estructura de control: *la iteración utiliza una estructura repetitiva y la recursión utiliza una estructura de selección*. Tanto la iteración como la recursión implican repetición: la iteración utiliza explícitamente una estructura repetitiva mientras que la recursión consigue la repetición mediante llamadas repetidas al método. La iteración y la recursión implican cada una un test de terminación *(condición de parada)*. La iteración termina cuando la condición del bucle no se cumple, mientras que la recursión termina cuando se reconoce un caso base o se alcanza la condición de parada.

La recursión tiene muchas desventajas. Se invoca repetidamente al mecanismo de llamadas a métodos y, en consecuencia, se necesita un tiempo suplementario para realizar cada llamada.

Esta característica puede resultar cara en tiempo de procesador y espacio de memoria. Cada llamada recursiva produce una nueva creación y copia de las variables de la función, esto consume más memoria e incrementa el tiempo de ejecución. Por el contrario, la iteración se produce dentro de un método, de modo que las operaciones suplementarias en la llamada al método y en la asignación de memoria adicional son omitidas.

Entonces, ¿cuáles son las razones para elegir la recursión? La razón fundamental es que existen numerosos problemas complejos que poseen naturaleza recursiva y, en consecuencia, son más fáciles de implementar con algoritmos de este tipo. Sin embargo, en condiciones críticas de tiempo y de memoria; es decir, cuando el consumo de tiempo y memoria sean decisivos o concluyentes para la resolución del problema, la solución a elegir debe ser, normalmente, la iterativa.

A tener en cuenta

Cualquier problema que se pueda resolver recursivamente tiene, al menos, una solución iterativa utilizando una pila. Un enfoque recursivo se elige, normalmente, con preferencia a un enfoque iterativo cuando resulta más natural para la resolución del problema y produce un programa más fácil de comprender y de depurar. Otra razón para elegir una solución recursiva es que una solución iterativa puede no ser clara ni evidente.

Consejo de programación

Se ha de evitar utilizar recursividad en situaciones de rendimiento crítico o exigencia de altas prestaciones en tiempo y en memoria, ya que las llamadas recursivas emplean tiempo y consumen memoria adicional. No es conveniente el uso de una llamada recursiva para sustituir un simple bucle.

Ejemplo 5.3

Dado un número natural n, obtener la suma de los dígitos de que consta. Presentar un algoritmo recursivo y otro iterativo.

El ejemplo ofrece una muestra clara de comparación entre la resolución de modo iterativo y de modo recursivo. Se asume que el número es natural y que, por tanto, no tiene signo. La suma de los dígitos se puede expresar:

```
suma = suma(n/10) + modulo (n,10)   para n > 9
suma = n                            para n ≤ 9, caso base
```

Para, por ejemplo, `n = 259`:

```
suma = suma(259/10)   +  modulo (259,10)        → 2 + 5 + 9 = 16
           ↓
suma = suma(25/10)    +  modulo (25,10)         → 2 + 5    ↑
           ↓
suma = suma(2/10)     +  modulo (2,10)          → 2 ↑
```

El caso base, el que se resuelve directamente, es `n ≤ 9` y, a su vez, es la condición de parada.

Solución recursiva

```
int sumaRecursiva(int n)
{
   if (n <= 9)
      return n;
   else
      return sumaRecursiva(n/10) + n%10;
}
```

Solución iterativa

La solución iterativa se construye con un bucle *mientras,* repitiendo la acumulación del resto de dividir `n` por `10` y actualizando `n` en el cociente. La condición de salida del bucle es que `n` sea menor o igual que `9`.

```
int sumaIterativa(int n)
{
   int suma = 0;
   while (n > 9)
   {
      suma += n%10;
      n /= 10;
   }
   return (suma+n);
}
```

5.3.1. Directrices en la toma de decisión iteración/recursión

1. Considérese una solución recursiva sólo cuando una solución iterativa *sencilla* no sea posible.
2. Utilícese una solución recursiva sólo cuando la ejecución y eficiencia de la memoria de la solución esté dentro de límites aceptables, considerando las limitaciones del sistema.
3. Si son posibles las dos soluciones, iterativa y recursiva, la solución recursiva siempre requerirá más tiempo y espacio debido a las llamadas adicionales a los métodos.
4. En ciertos problemas, la recursión conduce a soluciones que son mucho más fáciles de leer y de comprender que su alternativa iterativa. En estos casos, los beneficios obtenidos con la claridad de la solución suelen compensar el coste extra (en tiempo y memoria) de la ejecución de un programa recursivo.

Consejo de programación

Un método recursivo que tiene la llamada recursiva como última sentencia *(recursión final)* puede transformarse fácilmente en iterativa reemplazando la llamada mediante un bucle condicional que chequee el caso base.

5.3.2. Recursión infinita

La iteración y la recursión pueden producirse infinitamente. Un bucle infinito ocurre si la prueba o test de continuación de bucle nunca se vuelve falsa; una recursión infinita ocurre si la etapa de recursión no reduce el problema en cada ocasión, de modo que converja sobre el caso base o condición de salida.

La **recursión infinita** significa que cada llamada recursiva produce otra llamada recursiva y ésta, a su vez, otra llamada recursiva, y así para siempre. En la práctica, dicho método se ejecutará hasta que la computadora agote la memoria disponible y se produzca una terminación anormal del programa.

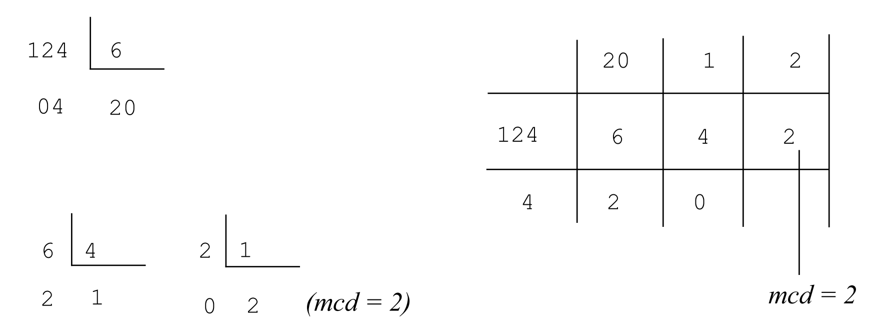
El flujo de control de un método recursivo requiere tres condiciones para una terminación normal:

- Un test para detener (o continuar) la recursión *(condición de salida o caso base).*
- Una llamada recursiva (para continuar la recursión).
- Un caso final para terminar la recursión.

Ejemplo 5.4

Deducir cual es la condición de salida del método `mcd()`, *que calcula el mayor denominador común de dos números enteros,* `b1` *y* `b2` *(el* **mcd,** *máximo común divisor, es el entero mayor que divide a ambos números).*

Supongamos dos número, `6` y `124`; el procedimiento clásico para hallar el *mcd* es la obtención de divisiones sucesivas entre ambos números (`124` entre `6`); si el resto no es `0`, se divide el número menor (`6`, en el ejemplo) por el resto (`4`, en el ejemplo), y así sucesivamente hasta que el resto sea `0`.



En el caso de `124` y `6`, donde el mcd es `2`, la condición de salida es que el resto sea cero. El algoritmo del *mcd* entre dos números *m* y *n* es:

- `mcd(m,n)` *es* `n` *si* `n <= m` *y* `n` *divide a* `m`
- `mcd(m,n)` *es* `mcd(n, m)` *si* `m < n`
- `mcd(m,n)` *es* `mcd(n, resto de m divido por n)` *en caso contrario.*

El método recursivo:

```
static int mcd(int m, int n)
{
   if (n <= m && m % n == 0)
      return n;
   else if (m < n)
      return mcd(n, m);
   else
      return mcd(n, m % n);
}
```

5.4. ALGORITMOS *DIVIDE Y VENCERÁS*

Una de las técnicas más importantes para la resolución de muchos problemas de computadora es la denominada *divide y vencerás*. El diseño de algoritmos basados en esta técnica consiste en transformar (dividir) un problema de tamaño *n* en problemas más pequeños, de tamaño menor que *n,* pero similares al problema original, de modo que resolviendo los subproblemas y combinando las soluciones se pueda construir fácilmente una solución del problema completo *(vencerás)*.

Normalmente, el proceso de división de un problema en otros de tamaño menor va a dar lugar a que se llegue al *caso base,* cuya solución es inmediata. A partir de la obtención de la solución del problema para el caso base, se combinan soluciones que amplían el tamaño del problema resuelto, hasta que el problema original queda también resuelto.

Por ejemplo, se plantea el problema de dibujar un segmento que está conectado por los puntos en el plano `(x1,y1)` y `(x2,y2)`. El problema puede descomponerse así: determinar el punto medio del segmento, dibujar dicho punto y *dibujar los dos segmentos mitad obtenidos al dividir el segmento original por el punto mitad*. El tamaño del problema se ha reducido a la mitad, el hecho de dibujar un segmento se ha transformado en dibujar dos segmentos con un tamaño de justamente la mitad. Sobre cada segmento mitad se vuelve aplicar el mismo procedimiento, de tal forma que llega un momento en que, a base de dividir el segmento, se alcanza uno de longitud cercana a cero, se ha llegado al caso base, y se dibuja un punto. Cada tarea realiza las mismas acciones, por lo que se puede plantear con llamadas recursivas al proceso de dibujar el segmento cada vez con un tamaño menor, exactamente la mitad.

Un algoritmo *divide y vencerás* se define de manera recursiva, de tal modo que se llama a sí mismo sobre un conjunto menor de elementos. Normalmente, se implementan con dos llamadas recursivas, cada una con un tamaño menor. Se alcanza el *caso base* cuando el problema se resuelve directamente.

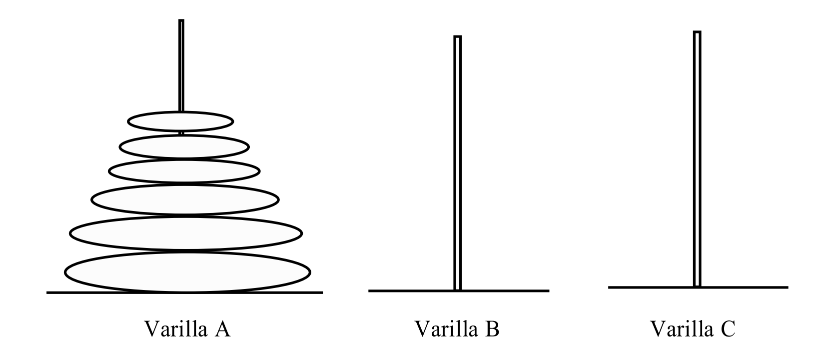
Norma

Un algoritmo *divide y vencerás* consta de dos partes. La primera, *divide recursivamente* el problema original en subproblemas cada vez mas pequeños. La segunda, soluciona *(vencerás)* el problema dando respuesta a los subproblemas. Desde el caso base se empieza a combinar soluciones de subproblemas hasta que queda resuelto el problema completo.

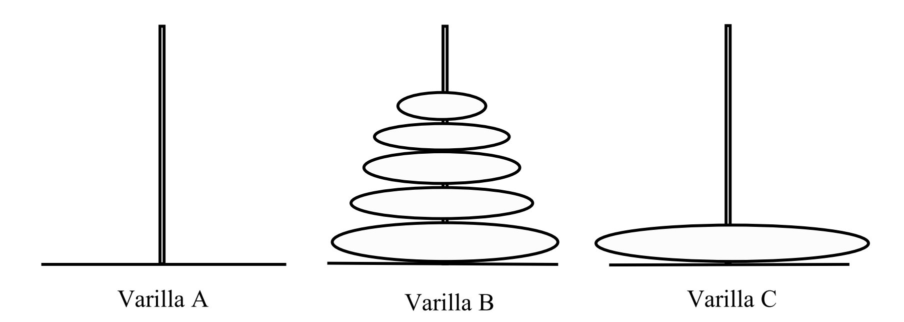
Problemas clásicos resueltos mediante recursividad son las Torres de Hanoi, el método de búsqueda binaria, la ordenación rápida, la ordenación por mezclas, etc.

5.4.1. Torres de Hanoi

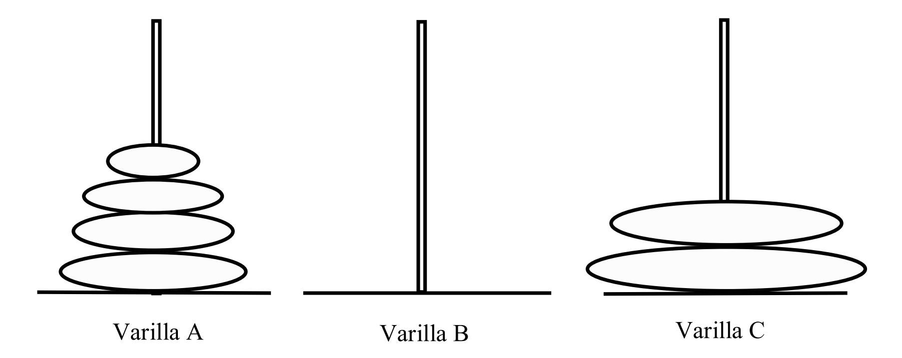
Este juego (un algoritmo clásico) tiene sus orígenes en la cultura oriental y en una leyenda sobre el Templo de Brahma, cuya estructura simulaba una plataforma metálica con tres varillas y discos en su interior. El problema en cuestión suponía la existencia de 3 varillas (A, B y C) o postes en los que se alojaban discos (*n* discos) que se podían trasladar de una varilla a otra libremente, pero con una condición: cada disco era ligeramente inferior en diámetro al que estaba justo debajo de él.



Se ilustra el problema con tres varillas con seis discos en la varilla A, y se desea trasladar a la varilla C conservando la condición de que cada disco sea ligeramente inferior en diámetro al que tiene situado debajo de él. Por ejemplo, se pueden cambiar cinco discos de golpe de la varilla A a la varilla B, y el disco más grande a la varilla C. Ya ha habido una transformación del problema en otro de menor tamaño, se ha divido el problema original.



Ahora el problema se centra en pasar los cinco discos de la varilla B a la varilla C. Se utiliza un método similar al anterior, pasar los cuatro discos superiores de la varilla B a la varilla A y, a continuación, se pasa el disco de mayor tamaño de la varilla B a la varilla C, y así sucesivamente. El proceso continúa del mismo modo, siempre dividiendo el problema en dos de menor tamaño, hasta que finalmente se queda un disco en la varilla B, que es *el caso base* y, a su vez, la condición de parada.



La solución del problema es claramente recursiva. Además, con las dos partes mencionadas anteriormente: división recursiva y solución a partir del caso base.

Diseño del algoritmo

El juego consta de tres varillas (alambres) denominadas *varilla inicial, varilla central* y *varilla final.*

En la varilla inicial se sitúan `n` discos que se apilan en orden creciente de tamaño con el mayor en la parte inferior. El objetivo del juego es mover los `n` discos desde la varilla inicial a la varilla final. Los discos se mueven uno a uno con la condición de que un disco mayor nunca puede ser situado encima de un disco más pequeño. El método `hanoi()` declara las varillas o postes como datos de tipo `char`. En la lista de parámetros, el orden de las variables es:

```
varinicial,  varcentral,  varfinal
```

e implica que se están moviendo discos desde la varilla inicial a la final utilizando la varilla central como auxiliar para almacenar los discos. Si $n = 1$ se tiene el caso base, se resuelve directamente moviendo el único disco desde la varilla inicial a la varilla final. El algoritmo es el siguiente:

1. **Si *n* es 1**
   - **1.1** Mover el disco `1` de `varinicial` a `varfinal`.
2. **Si no**
   - **1.2** Mover `n - 1` discos desde `varinicial` hasta `varcentral` utilizando `varfinal` como auxiliar.
   - **1.3** Mover el disco `n` desde `varinicial` a `varfinal`.
   - **1.4** Mover `n - 1` discos desde `varcentral` a `varfinal` utilizando como auxiliar `varinicial`.

Es decir, si `n` es 1, se alcanza *el caso base,* la condición de salida o terminación del algoritmo. Si `n` es mayor que 1, las etapas recursivas `1.2, 1.3 y 1.4` son tres subproblemas más pequeños, uno de los cuales es la condición de salida.

Las figuras 5.1 a 5.6 muestran el algoritmo anterior:

Etapa 1: Mover *n-1* discos desde varilla inicial (A).

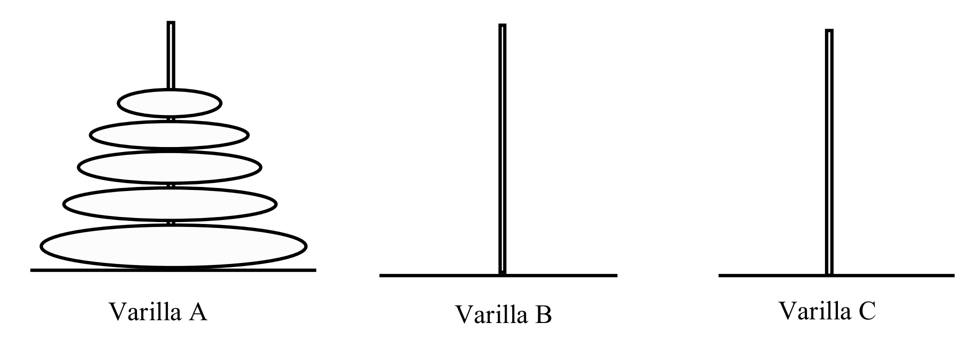


Figura 5.1 Situación inicial

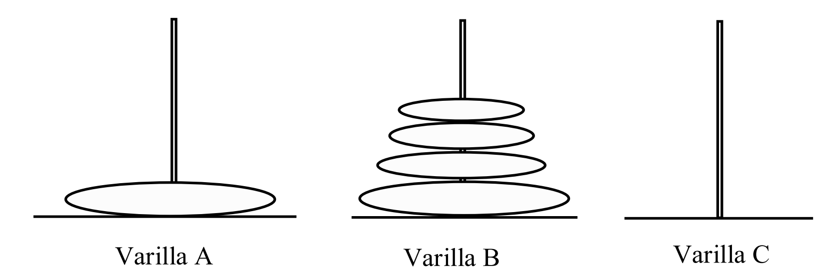


Figura 5.2 Después del movimiento

Etapa 2: Mover un disco desde A a C.

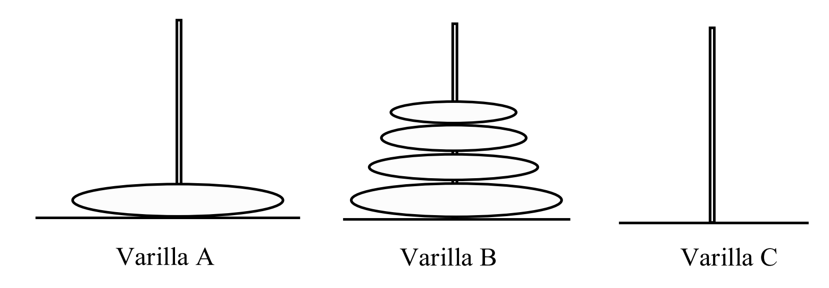


Figura 5.3 Situación de partida de etapa 2

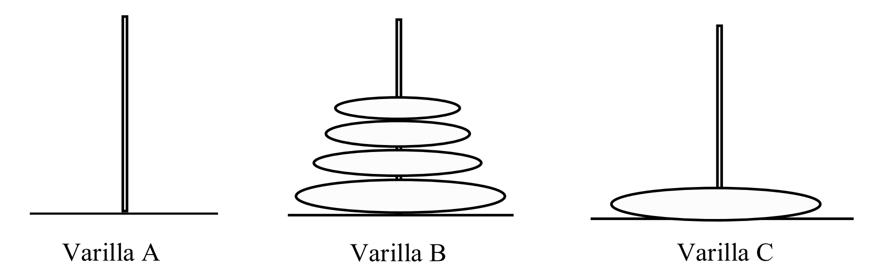


Figura 5.4 Después de la etapa 2

Etapa 3: Mover discos desde B a C.

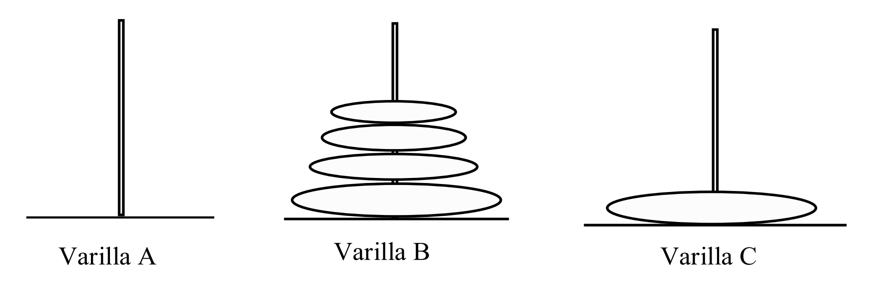


Figura 5.5 Antes de la etapa 3

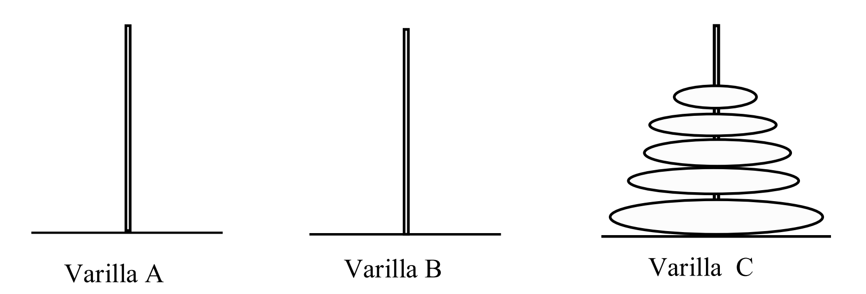


Figura 5.6 Después de la etapa 3

La primera etapa en el algoritmo mueve *n - 1* discos desde la varilla inicial a la varilla central utilizando la varilla final como almacenamiento temporal. Por consiguiente, el orden de parámetros en la llamada recursiva es `varinicial, varfinal` y `varcentral`.

```
hanoi(varinicial, varfinal, varcentral, n-1);
```

La segunda etapa, simplemente mueve el disco mayor desde la varilla inicial a la varilla final:

```
System.out.println("Mover disco " + n + " desde varilla " + varinicial
                        + " a varilla " + varfinal);
```

La tercera etapa del algoritmo mueve *n - 1* discos desde la varilla central a la varilla final utilizando `varinicial` para almacenamiento temporal. Por consiguiente, el orden de parámetros en la llamada al método recursivo es: *`varcentral, varinicial y varfinal.`*

```
hanoi(varcentral, varinicial, varfinal, n-1);
```

Implementación de las Torres de Hanoi

La implementación del algoritmo se apoya en los nombres de las tres varillas: `'A'`, `'B'` y `'C'`, que se pasan como parámetros al método `hanoi()`. El método tiene un cuarto parámetro, que es el número de discos, `n`, que intervienen. Se obtiene un listado de los movimientos que transferirán los `n` discos desde la varilla inicial, `'A'`, a la varilla final, `'C'`. La codificación es:

```
static
void hanoi(char varinicial, char varcentral, char varfinal, int n)
{
   if ( n == 1)
      System.out.println("Mover disco " + n + " desde varilla " +
                            varinicial + " a varilla " + varfinal);
   else
   {
      hanoi(varinicial, varfinal, varcentral, n-1);
      System.out.println("Mover disco " + n + " desde varilla " +
                           varinicial + " a varilla " + varfinal);

      hanoi(varcentral, varinicial, varfinal, n - 1);
   }
}
```

Análisis del algoritmo Torres de Hanoi

Fácilmente se puede encontrar el árbol de llamadas recursivas para `n = 3` discos, que en total realiza $7 = (2^3 - 1)$ llamadas a `hanoi()` y escribe `7` movimientos de disco. El problema de `5` discos se resuelve con $31 = (2^5 - 1)$ llamadas y `31` movimientos. Si se supone que `T(n)` es el número de movimientos para `n` discos, teniendo en cuenta que el método realiza dos llamadas con `n-1` discos y mueve el disco `n`, se tiene la recurrencia:

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ 2T(n-1) + 1 & \text{si } n > 1 \end{cases}$$

Los sucesivos valores que toma `T`, según `n`: `1, 3, 7, 15, 31, 63 ...` $2^n - 1$.

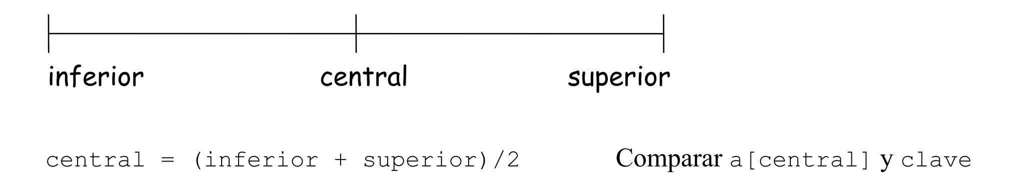
En general, el número de movimientos requeridos para resolver el problema de *n* discos es $2^n - 1$. Cada llamada al método requiere la asignación e inicialización de un área local de datos en la memoria; por ello, el tiempo de computadora se incrementa exponencialmente con el tamaño del problema.

> ***Nota de ejecución***
>
> La complejidad del algoritmo que resuelve el problema de las Torres de Hanoi es exponencial. Por consiguiente, a medida que crece n, aumenta exponencialmente el tiempo de ejecución de la función.

5.4.2. Búsqueda binaria

La búsqueda binaria es un método de localización de una clave especificada dentro de una lista o array ordenado de *n* elementos que realiza una exploración de la lista hasta que se encuentra o se decide que no se está en la lista. El algoritmo de búsqueda binaria se puede describir recursívamente aplicando la técnica *divide y vencerás*.

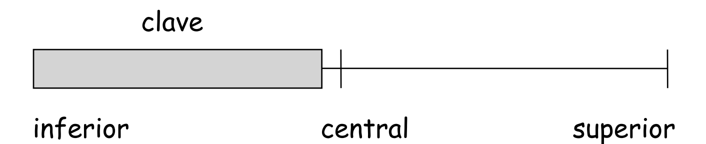
Se tiene una lista ordenada `a[]` con un límite inferior y un límite superior. Dada una clave (valor buscado), comienza la búsqueda en la posición central de la lista (índice central).



Si hay coincidencia (se encuentra la clave), se tiene la condición de terminación que permite detener la búsqueda y devolver el índice central. En caso contrario (no se encuentra la clave), dado que la lista está ordenada, se centra la búsqueda en la "sublista inferior" (a la izquierda de la posición central) o en la "sublista superior" (a la derecha de la posición central). El problema de la búsqueda se ha dividido en la mitad, el tamaño de la lista donde buscar es la mitad del

tamaño anterior.El tamaño de la lista se reduce cada vez a la mitad, así hasta que se encuentre el elemento, o bien la lista resultante esté vacía.

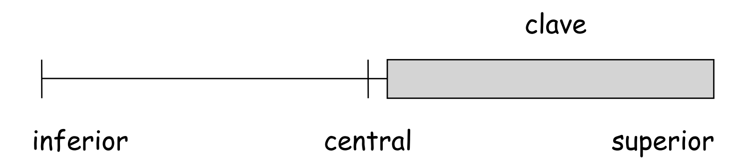
1. Si `clave < a[central]`, el valor buscado sólo puede estar en la mitad izquierda de la lista con elementos en el rango, `inferior` a `central - 1`.



Búsqueda en sublista izquierda

`[inferior .. central - 1]`

2. Si `clave > a[central]`, el valor buscado sólo puede estar en la mitad superior de la lista con elementos en el rango de índices, `central + 1` a `superior`.



3. La búsqueda continúa en sublistas más y más pequeñas, exactamente la mitad, con dos llamada recursivas: una se corresponde con la sublista inferior y la otra, con la sublista superior. El algoritmo termina con éxito *(aparece la clave buscada)* o sin éxito *(no aparece la clave buscada)*, situación que ocurrirá cuando el límite superior de la lista sea más pequeño que el límite inferior. La condición `inferior > superior` es la condición de salida o terminación sin éxito, y el algoritmo devuelve el índice -1.

Se codifica en un método `static` que formaría parte de una clase de utilidades.

```
static int busquedaBR(double a[], double clave,
                      int inferior, int superior)
{
   int central;
   if (inferior > superior) // no encontrado
      return -1;
   else
   {
      central = (inferior + superior)/2;
      if (a[central] == clave)
         return central;
      else if (a[central] < clave)
         return busquedaBR(a, clave, central+1, superior);
      else
         return busquedaBR(a, clave, inferior, central-1);
   }
}
```

Análisis del algoritmo

El peor de los casos que hay que contemplar en una búsqueda es que ésta no tenga éxito. El tiempo del algoritmo recursivo de búsqueda binaria depende del número de llamadas. Cada llamada reduce la lista a la mitad, y progresivamente se llega a que el tamaño de la lista es unitario; en la siguiente llamada, el tamaño es `0 (inferior > superior)` y termina el algoritmo. La secuencia siguiente describe los sucesivos tamaños:

$$n/2,\ n/2^2,\ n/2^3,\ \ldots\ ,\ n/2^t = 1$$

tomando logaritmo, $t = \log n$. Por tanto, el número de llamadas es $\lfloor \log n \rceil + 1$. Cada llamada es de complejidad constante, así que se puede afirmar que la complejidad, en término de notación `O`, es logarítmica `O(log n)`.

5.5. *BACKTRACKING*, ALGORITMOS DE VUELTA ATRÁS

La resolución de algunos problemas exige probar sistemáticamente todas las posibilidades que pueden existir para encontrar una solución. Los algoritmos de *vuelta atrás* utilizan la recursividad para probar cada una de las posibilidades de encontrar la solución.

Este método de resolución de problemas recurre a realizar una búsqueda exhaustiva, sistemática, de una posible solución al problema planteado. Descompone el proceso de búsqueda o tanteo de una solución en tareas parciales, cada una de las cuales realiza las mismas acciones que la tarea anterior, por eso se expresa, frecuentemente, en forma recursiva.

Por ejemplo, el problema de determinar los sucesivos saltos que debe de hacer un caballo de ajedrez, desde una posición inicial cualquiera (con su forma típica de moverse) para que pase por todas las casillas de un tablero vacío; la tarea parcial es realizar un salto válido, en cuanto que dicho salto esté dentro de las coordenadas del tablero y no haya pasado ya por la casilla destino. En este problema, que se resolverá posteriormente, todas las posibilidades son los ocho posibles saltos que el caballo puede realizar desde una casilla dada.

El proceso general de los algoritmos de *vuelta atrás* se contempla como un método de prueba o búsqueda, que gradualmente construye tareas básicas y las inspecciona para determinar si conducen a la solución del problema. Si una tarea no conduce a la solución, prueba con otra tarea básica hasta agotar todas las posibilidades. Es una prueba sistemática hasta llegar a la solución, o bien determinar que no hay solución por haberse agotado todas las opciones que probar.

> ***Nota de ejecución***
>
> Una de las características principales de los algoritmos de *vuelta atrás* es la búsqueda exhaustiva, con todas las posibilidades, de soluciones parciales que conduzcan a la solución del problema. Otra característica es la *vuelta atrás*, en el sentido de que si una solución o tarea parcial no conduce a la solución global del problema se *vuelve atrás* para probar (ensayar) con otra de las posibilidades de solución parcial.

Modelo de los algoritmos de vuelta atrás

El esquema general de la técnica de resolución de problemas *backtracking*:

```
procedimiento ensayarSolucion
inicio
```

```
    <inicializar cuenta de posibles selecciones>
    repetir
       <tomar siguiente selección (tarea)>
       <determinar si selección es valida>
       si válido entonces
          <anotar selección>
          si <problema solucionado>  entonces
             exito = true
          si no
             ensayarSolución {llamada para realizar otra tarea}
                {vuelta atrás, se analiza si se ha alcanzado la solución del problema}
             si no exito entonces
                <borrar anotación> {el bucle se encarga de probar con otra selección}
             fin _ si
          fin _ si
       fin _ si
    hasta (exito) o (<no más posibilidades>)
 fin
```

Este esquema tendrá las variaciones necesarias para adaptarlo a la casuística del problema a resolver. A continuación, se aplica esta forma de resolución a diversos problemas, como el del *salto del caballo* y el de las *ocho reinas.*

5.5.1. Problema del Salto del caballo

En un tablero de ajedrez de `n x n` casillas, se tiene un caballo situado en la posición inicial de coordenadas (x_0,y_0). El problema consiste en encontrar, si existe, un circuito que permita al caballo pasar exactamente una vez por cada una de las casillas de tablero, teniendo en cuenta los movimientos (saltos) permitidos a un caballo de ajedrez.

Este es un ejemplo clásico de problema que se resuelve con el esquema del algoritmo de *vuelta atrás.* El problema consiste en buscar la secuencia de saltos que tiene que dar el caballo, partiendo de una casilla cualquiera, para pasar por cada una de las casillas del tablero. Se da por supuesto que el tablero está vacío, no hay figuras excepto el caballo. Lo primero que hay que tener en cuenta es que el caballo, desde una casilla, puede realizar hasta 8 movimientos, como muestra la Figura 5.7.

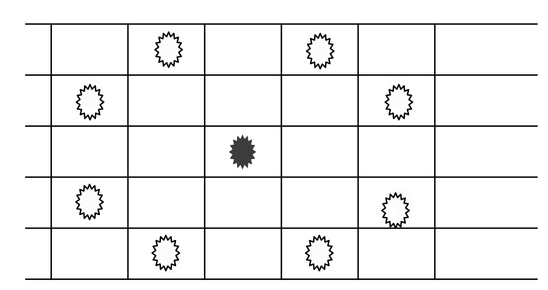


Figura 5.7 Los ocho posibles saltos del caballo

Por consiguiente, el número de *posibles selecciones* en este problema es ocho. La tarea básica, anteriormente se ha denominado *solución parcial*, en que se basa el problema consiste en que el caballo realice un nuevo movimiento entre los ocho posibles.

Los ocho posibles movimientos de caballo se obtienen sumando a su posición actual, `(x,y)`, unos desplazamientos relativos, que son:

```
d = {(2,1), (1,2), (-1,2), (-2,1), (-2,-1), (-1,-2), (1,-2), (2,-1)}
```

Por ejemplo, si el caballo se encuentra en la casilla `(3,5)`, los posibles movimientos que puede realizar:

```
{(5,6), (4,7), (2,7), (1,6), (1,4), (2,3), (4,3), (5,4)}
```

No siempre será posible realizar los ocho movimientos, se debe comprobar que la casilla destino esté dentro del tablero y también que el caballo no haya pasado previamente por ella. En caso de ser posible el movimiento, se anota, guardando el número del salto realizado.

```
tablero[x][y] = numeroSalto {número del Salto}
nuevaCoorX    = x + d[k][1]
nuevaCoorY    = y + d[k][2]
```

Las llamadas a la función que resuelve el problema transmiten las nuevas coordenadas y el nuevo salto a realizar son:

```
saltoCaballo(nuevaCoorX, nuevaCoorY, numeroSalto+1)
```

La condición que determina que el problema se ha resuelto está ligada con el objetivo que se persigue, y en este problema es que se haya pasado por las n^2 casillas; en definitiva, que el caballo haya realizado $n^2 - 1$ (63) saltos. En ese momento, se pone a `true` la variable `exito`.

¿Qué ocurre si se agotan los ocho posibles movimientos sin alcanzar la solución? Se vuelve al movimiento anterior, *vuelta atrás,* se borra la anotación para ensayar con el siguiente movimiento. Y si también se han agotado los movimientos, ocurre lo mismo: se vuelve al que fue su movimiento anterior para ensayar, si es posible, con el siguiente movimiento de los ocho posibles.

Representación del problema

De manera natural, el tablero se representa mediante una matriz de enteros para guardar el número de salto en el que pasa el caballo. En Java, los arrays siempre tienen como índice inferior 0, se ha preferido reservar una fila y una columna más para el tablero y así representarlo más fielmente.

```
final int N = 8;
final int n = (N+1);

int [][] tablero = new int[n][n];
```

Una posición del tablero puede contener:

$$\texttt{tablero[x,y]} = \begin{cases} 0 & \text{por la casilla } (x,y) \text{ no pasó el caballo.} \\ i & \text{por la casilla } (x,y) \text{ pasó el caballo en el salto } i. \end{cases}$$

Los desplazamientos relativos que determinan el siguiente salto del caballo se guardan en una matriz constante con los valores predeterminados.

Parámetros de la llamada recursiva

Los únicos parámetros que va a tener el método recursivo que resuelve el problema son los necesarios para que el caballo realice un nuevo movimiento, que son las coordenadas actuales y el número de salto del caballo. El *flag* que indica si se ha completado el problema será una variable de la clase en la que se implementa el problema.

Codificación del algoritmo *Salto del caballo*

En la clase *CaballoSaltador* se declaran los atributos que representan el tamaño y el tablero (matriz). También se define el atributo `exito`, de tipo lógico, que será puesto a *verdadero* si el método recursivo que resuelve el problema encuentra una solución. Además, la matriz `SALTO` guarda los ocho desplazamientos relativos a una posición dada del caballo.

El método interno `saltoDelCaballo()` realiza todo el trabajo mediante llamadas recursivas; el método *público* `resolverProblema()` llama a `saltoCaballo()` con las coordenadas iniciales que se establecen en el constructor. Éste valida que las coordenadas estén dentro del tablero, inicializa el tablero y establece el valor de `exito` a *falso*.

```
class CaballoSaltador
{
   static final int N = 8;
   static final int n = (N+1);
   private int [][] tablero = new int[n][n];
   private boolean exito;
   private int [][]SALTO = {{2,1}, {1,2}, {-1,2}, {-2,1},
                            {-2,-1}, {-1,-2}, {1,-2}, {2,-1}};
   private int x0, y0;
      // constructor
   public CaballoSaltador(int x, int y) throws Exception
   {
      if ((x < 1) || (x > N) ||
         (y < 1) || (y > N))
         throw new Exception("Coordenadas fuera de rango");
      x0 = x;
      y0 = y;
      for(int i = 1; i<= N; i++)
         for(int j = 1; j<= N; j++)
            tablero[i][j] = 0;
      tablero[x0][y0] = 1;
      exito = false;
   }
   public boolean resolverProblema()
   {
      saltoCaballo(x0, y0, 2);
      return exito;
   }
   private void saltoCaballo(int x, int y, int i)
   {
      int nx, ny;
      int k;
      k = 0; // inicializa el conjunto de posibles movimientos
      do {
         k++;
         nx = x + SALTO[k-1][0];
```

```
            ny = y + SALTO[k-1][1];
            // determina si nuevas coordenadas son aceptables
            if ((nx >= 1) && (nx <= N) && (ny >= 1) && (ny <= N)
                           &&
                (tablero[nx][ny] == 0))
            {
               tablero[nx][ny]= i; // anota movimiento
               if (i < N*N)
               {
                  saltoCaballo(nx, ny, i+1);
                  // se analiza si se ha completado la solución
                  if (!exito)
                  { // no se alcanzó la solución
                     tablero[nx][ny] = 0;  // se borra anotación
                  }
               }
               else
                  exito = true;   // tablero cubierto
            }
         } while ((k < 8) && !exito);
      }
      //muestra por pantalla los pasos del caballo
      void escribirTablero()
      {
         for(int i = 1; i <= N; i++)
         {
            for(int j = 1; j <= N; j++)
               System.out.print(tablero[i][j] + "  ");
            System.out.println();
         }
      }
   }
```

Ejercicio 5.3

Escribir una aplicación que, a partir de una casilla inicial del caballo, resuelva el problema del salto del caballo.

La aplicación lee las coordenadas iniciales, crea un objeto de la clase `CaballoSaltador` y llama al método `resolverProblema()`. Se escribe el contenido del tablero si ha resuelto el problema.

```
import java.io.*;

public class Caballo
{
   public static void main(String[] ar)
   {
      int x, y;
      boolean solucion;
      BufferedReader entrada = new BufferedReader(
                           new InputStreamReader(System.in));
      try {
         System.out.println("Posicion inicial del caballo. ");
         System.out.print(" x = ");
```

```
            x = Integer.parseInt(entrada.readLine());
            System.out.print(" y = ");
            y = Integer.parseInt(entrada.readLine());
            CaballoSaltador miCaballo = new CaballoSaltador(x,y);
            solucion = miCaballo.resolverProblema();
            if (solucion)
               miCaballo.escribirTablero();
         }
         catch (Exception m)
         {
            System.out.println("No se pudo probar el algoritmo, " + m);
         }
      }
   }
```

Nota de ejecución

Los algoritmos que hacen una búsqueda exhaustiva de la solución tienen tiempos de ejecución muy elevados. La complejidad es exponencial, puede ocurrir, que si no se ejecuta en un ordenador potente, este se bloquee.

5.5.2. Problema de las ocho reinas

El problema se plantea de la forma siguiente: *dado un tablero de ajedrez (8 x 8 casillas), hay que situar ocho reinas de forma que ninguna de ellas pueda actuar ("comer") sobre ninguna de las otras.* Éste es otro de los ejemplos del uso de los métodos de búsqueda sistemática y de los algoritmos de *vuelta atrás*. Hay que recordar, en primer lugar, la regla del ajedrez respecto de los movimientos de la reina. Ésta puede moverse a lo largo de la columna, fila y diagonales de la casilla donde se encuentra.

Antes de empezar a aplicar los pasos que siguen estos algoritmos, se van a *podar* posibles movimientos de las reinas. Cada columna puede contener una y sólo una reina, la razón de esta restricción es inmediata: si en la columna 1 se encuentra la reina 1, y en esta columna se sitúa otra reina, entonces se *atacan* mutuamente. De las `n x n` casillas que puede ocupar una reina, se limita su ubicación a las 8 casillas de la columna en la que se encuentra, de tal forma que si se numeran las reinas de 1 a 8, entonces la reina `i` se sitúa en alguna de las casillas de la columna `i`.

Lo primero a considerar, a la hora de aplicar el algoritmo de vuelta atrás, es la *tarea básica* que exhaustivamente se realiza; la propia naturaleza del problema de las 8 reinas determina que la tarea sea colocar la reina `i`, *tantear* si es posible ubicar la reina número `i`, para lo que hay 8 alternativas, que son las 8 posibles filas de la columna `i`. La comprobación de que una selección es válida tiene que hacerse investigando que en la fila seleccionada y en las dos diagonales en las que una reina puede atacar no haya otra reina colocada anteriormente. En cada paso se amplía la solución parcial al problema, ya que aumenta el número de reinas colocadas.

La segunda cuestión es analizar la *vuelta atrás*. ¿Qué ocurre si no se alcanza la solución, si no se es capaz de colocar las 8 reinas? Se borra la ubicación de la reina, la fila donde se ha colocado, y se ensaya con la siguiente fila válida.

La realización de cada tarea supone ampliar el número de reinas colocadas hasta llegar a la solución completa, o bien determinar que no es posible colocar la reina actual en las 8 posibles filas. Entonces, en la *vuelta atrás* se coloca la reina actual en otra fila válida para realizar un nuevo *tanteo*.

Representación del problema

En cuanto a los tipos de datos para representar las reinas en el tablero, hay que tener en cuenta que éstos permitan realizar las acciones de verificar que *no se coman* las reinas lo más eficientemente posible.

Debido a que el objetivo del problema es encontrar la fila en la que se sitúa la reina de la columna `i`, se define el array entero, `reinas[]`, de tal forma que cada elemento contenga el índice de fila donde se sitúa la reina, o bien cero. El número de reina, `i`, es a su vez el índice de columna dentro de la cual se puede colocar entre los ocho posibles valores de fila.

El array que se define tiene una posición más, y así la numeración de las reinas coincide con el índice del array.

```
final int N = 8;
final int n = (N+1);
int [] reinas = new int[n];
```

Verificación de la ubicación de una reina

Al colocar la reina `i` en la fila `j` hay que comprobar que no se ataque con las reinas colocadas anteriormente. Un reina `i` situada en la fila `j` de la columna `i` ataca, o es atacada, por otra reina que esté en la misma fila, o bien por otra que esté en la misma diagonal `(i-j)`, o en la diagonal `(i+j)`. El método `valido()` realiza esta comprobación mediante un bucle con tantas iteraciones como reinas situadas actualmente.

Parámetros de la llamada recursiva

El método recursivo tiene los parámetros necesarios para situar una nueva reina. En este problema sólo es necesario el número de reina que se va a colocar; el array con las reinas colocadas actualmente es un atributo de la clase, al igual que la variable lógica que se pone a *verdadero* si han sido puestas todas las reinas.

Codificación del algoritmo de las 8 reinas

En la clase `OchoReinas` se declaran como atributos los datos que se han establecido para representar el problema. El método `solucionReinas()` es la interfaz de la clase para resolver el problema. Este llama a `ponerReina()` que realiza todo el trabajo de resolución con llamadas recursivas.

```
public boolean solucionReinas()
{
   solucion = false;
   ponerReina(1);
   return solucion;
}
private void ponerReina(int i)
{
   int j;
   j = 0; // inicializa posibles movimientos
   do {
      j++;
      reinas[i] = j; // prueba a colocar reina i en fila j,
                     // a la vez queda anotado el movimiento
      if (valido(i))
      {
         if (i < N) // no completado el problema
```

```
        {
           ponerReina(i+1);
           // vuelta atrás
           if (!solucion)
              reinas[i] = 0;
        }
        else // todas las reinas colocadas
           solucion = true;
     }
  } while(!solucion && (j < 8));
}
private boolean valido(int i)
{
     /* Inspecciona si la reina de la columna i es atacada por
        alguna reina colocada anteriormente */
  int r;
  boolean libre;
  libre = true;
  for (r = 1; r <= i-1; r++)
  {
     // no esté en la misma fila
     libre = libre && (reinas[i] != reinas[r]);
     // no esté en alguna de las dos diagonales
     libre = libre && ((i + reinas[i]) != (r + reinas[r]));
     libre = libre && ((i - reinas[i]) != (r - reinas[r]));
  }
  return libre;
}
```

5.6. SELECCIÓN ÓPTIMA

En los problemas del *Salto de caballo* y *Ocho reinas* se ha aplicado la estrategia de *vuelta atrás* para encontrar una solución del problema y, a continuación, dejar de hacer llamadas recursivas. Para resolver el problema de encontrar las variaciones se ha seguido una estrategia similar, con la diferencia de que no se detiene al encontrar una variación, sino que sigue haciendo llamadas hasta *agotar todas las posibilidades,* todos los elementos.

Por consiguiente, un sencillo cambio al esquema algorítmico permite encontrar *todas las soluciones* a los problemas planteados; todo consiste en que el algoritmo muestre la solución encontrada en lugar de activar una variable `boolean` y siempre realice *la vuelta atrás, desanotando lo anotado* antes de hacer la llamada recursiva. Por ejemplo, el problema de las *Ocho reinas* encuentra una solución, pero realizando el cambio indicado se encuentran todas las formas de colocar la reinas en el tablero sin que se "coman".

Ejercicio 5.4

Dado un conjunto de pesos, se quieren escribir todas las combinaciones de ellos que totalicen un peso dado.

El problema se resuelve probando con todas las combinaciones posibles de los objetos disponibles. Cada vez que la suma de los pesos de una combinación sea igual al peso dado, se escribe en los objetos de los que consta. El problema se resuelve modificando el método `seleccion()`de

la clase `SeleccionObjeto`[1]. El atributo `encontrado` ya no es necesario, el bucle tiene que iterar con todos los objetos, cada vez que `suma` sea igual al peso objetivo, `objetivo`, se llama a `escribirSeleccion()`. Ahora, en la *vuelta atrás* siempre se borra de la bolsa la anotación del objeto realizada, para así probar con los `n` objetos.

```
public void seleccion(int candidato, int suma)
{

   while (candidato < n)
   {
      candidato++;
      if ((suma + objs[candidato-1]) <= objetivo)
      {
         k++; // es anotado
         bolsa[k-1] = candidato - 1;
         suma += objs[candidato-1];
         if (suma < objetivo)    // ensaya con siguiente objeto
         {
            seleccion(candidato, suma);
         }
         else // objetos totalizan el objetivo
            escribirSeleccion();
         // se borra la anotación
          k--;
         suma -= objs[candidato-1];
      }
   }
}
```

Los problemas que se adaptan al esquema de *selección óptima* no persiguen encontrar una situación fija o un valor predeterminado, sino averiguar, del conjunto de todas las soluciones, la óptima según unas condiciones que establecen qué es lo óptimo. Por tanto, hay que probar con todas las posibilidades que existen de realizar una nueva tarea para encontrar entre todas las soluciones la que cumpla una segunda condición o requisito, la *solución óptima*.

> ***Nota***
>
> La selección óptima implica probar con todas los posibles elementos de que se disponga para así de entre todas las configuraciones que cumplan una primera condición, seleccionar la más próxima a una segunda condición, que será la selección óptima.

El esquema general del algoritmo para encontrar la *selección óptima* es:

```
procedimiento ensayarSeleccion(objeto i)
inicio
   <inicializar cuenta de posibles selecciones>
   repetir
      <tomar siguiente selección (tarea)>
      <determinar si selección es valida>
```

[1] Consultar anexo en la web: Resolución de problemas con algoritmos de vuelta atrás.

```
        si válido entonces
           <anotar selección  >
           si  <es solución>  entonces
              si <mejor(solución)> entonces
                 <guardar selección>
              fin _ si
           fin _ si
            ensayarSeleccion(objeto i+1)
           <borrar anotación> {el bucle se encarga de probar con otra
                                  selección }
        fin _ si
     hasta (<no más posibilidades>)
  fin
```

Nota de eficiencia

El tiempo de ejecución de estos algoritmos crece muy rápidamente, y el número de llamadas recursivas crece exponencialmente (*complejidad exponencial)*. Por ello, es importante considerar el hecho de evitar una llamada recursiva si se sabe que no va a mejorar la actual selección óptima *(poda de las ramas en el árbol de llamadas)*.

5.6.1. Problema del viajante

El ejemplo del problema del viajante es el que mejor explica la *selección óptima: el viajante (piense en un representante de joyería) tiene que confeccionar la maleta, seleccionando entre n artículos aquellos cuyo valor sea máximo (lo óptimo es que la suma de valores sea máximo) y su peso no exceda de una cantidad, la que puede sustentar la maleta.*

Por ejemplo, la maleta de seguridad para llevar joyas es capaz de almacenar 1,5 kg y se tienen los objetos, representados por el par `(peso, valor)`: `(115 g, 100€), (90 g, 110€), (50 g, 60€), (120 g, 110€)`... Se pretende hacer una selección que no sobrepase los 1,5 kg y que la suma del valor asociado a cada objeto sea el máximo.

Para resolver el problema se generan todas las selecciones posibles con los `n` objetos disponibles. Cada selección debe cumplir la condición de no superar el peso máximo prefijado. Cada vez que se alcance una selección, se pregunta si es mejor que cualquiera de las anteriores y, en caso positivo, se guarda como la actual *mejor selección*. En definitiva, consiste en una *búsqueda exhaustiva*, un *tanteo sistemático* con los `n` objetos del problema; cada tanteo realiza la tarea de probar si incluir el objeto `i` va a dar lugar a una mejor selección, y también si la exclusión del objeto `i` dará lugar a una *mejor selección*.

```
  si solucion entonces
     si mejor(solucion) entonces
        optimo = solucion
```

Tarea básica

El objetivo del problema es que el valor que se pueda conseguir sea máximo; por esa razón, se considera que el valor más alto que se puede alcanzar es la suma de los valores de cada objeto; posteriormente, al excluir un objeto de la selección se ha de restar el valor que tiene.

La tarea básica en esta *búsqueda sistemática* es investigar si un objeto `i` es adecuado incluirlo en la selección actual. Será adecuado si el peso acumulado más el del objeto `i` no supera al peso de la maleta. En el caso de que sea necesario excluir el objeto `i` (*vuelta atrás* de las llamadas recursivas) de la selección actual, el criterio para seguir con el proceso de selección es que el valor total todavía alcanzable, después de esta exclusión, no sea menor que el valor óptimo (máximo) encontrado hasta ahora. La inclusión de un objeto supone un incremento del peso de la selección actual con el peso del objeto. A su vez, excluir un objeto de la selección supone que el valor que puede alcanzar la selección tiene que ser decrementado en el valor del objeto excluido. Cada tarea realiza las mismas acciones que la tarea anterior, por ello se expresa recursívamente.

La prueba de si es mejor selección se hace ensayando con todos los objetos disponibles, una vez que se alcance el último, el objeto `n`, es cuando se determina si el valor asociado a la selección es el mejor, el óptimo.

> ***Norma***
>
> En el proceso de *selección óptima* se prueba con la inclusión de un objeto `i` y con la exclusión de `i`. Para hacer más eficiente el algoritmo, se hace una *poda* de aquellas selecciones que no van a ser mejores; por ello, antes de probar con la exclusión se determina si la selección en curso puede alcanzar un mejor valor; si no es así ¿para qué ir por esa rama si no se va a conseguir una mejor selección?.

Representación de los datos

Se supone que el viajante tiene `n` objetos. Cada objeto tiene asociado el par `<peso, valor>`, por ello sendos arrays almacenan los datos de los objetos. La selección actual y la óptima van a tratarse como dos conjuntos de objetos; realmente, como la característica de cada objeto es el índice en el array de pesos y valores, los conjuntos incluirán únicamente el índice del objeto. Cada conjunto se representa por un atributo entero, el cardinal, y un array de índices.

Codificación

La clase `Optima` agrupa la representación y el algoritmo recursivo `seleccionOptima()`. El valor máximo que pueden alcanzar los objetos es la suma de los valores de cada uno, está representado por el atributo de la clase `totalValor`. El valor óptimo alcanzado en el proceso transcurrido está en el atributo `mejorValor`. Se inicializa al valor más bajo alcanzable, cero, así al menos habrá una selección que supere a dicho valor.

Los parámetros del método recursivo son los necesarios para realizar una nueva tarea: `i`, número del objeto a probar su inclusión; `pt`, peso de la selección actual y `va`, valor máximo alcanzable por la selección actual.

Ejercicio 5.5

Escribir la clase `Optima` *y una aplicación que solicite como datos de entrada los objetos (valor, peso) y el peso máximo que puede llevar el viajante.*

El método que resuelve el problema va a estar *oculto*, privado, de tal forma que la interfaz, `selecionOptima()`, inicializa los dos conjuntos, que representan a la selección actual y la óptima a *conjuntoVacio,* simplemente poniendo el cardinal a cero. En este método se suman los

valores asociados a cada objeto, que es el máximo valor alcanzable, y se asigna a `totalValor`. También se pone a cero el atributo `mejorValor` y llama al método recursivo.

```
class Optima
{
   private int n;
   private int []pesoObjs;
   private int []valorObjs;
   private int cardinalActual;
   private int [] actual;
   private int cardinalOptimo;
   private int [] optimo;
   private int pesoMaximo;
   private int mejorValor;
   // constructor, leer n y los objetos
   public Optima()
   {
      // realizar la entrada de los objetos
   }
   public void seleccionOptima()
   {
      int totalValor = 0;
      actual = new int[n];
      optimo = new int[n];
      mejorValor = 0;
      cardinalActual = 0; cardinalOptimo = 0
      for (int j = 0; j < n; j++)
         totalValor += valorObjs[j];
      seleccionOptima(1, 0, totalValor);
      escribirSeleccion();
   }
   private void seleccionOptima(int i, int pt, int va)
   {
      int valExclusion;
      if (pt + pesoObjs[i-1] <= pesoMaximo) // objeto i se incluye
      {
         cardinalActual++;
         actual[cardinalActual-1] = i-1; // índices del objeto
         if (i < n)
            seleccionOptima(i+1, pt + pesoObjs[i-1], va);
         else // los n objetos probados
            if (va > mejorValor) // nuevo optimo
            {
               mejorValor = va;
               System.arraycopy(actual,0,optimo,0,cardinalActual);
               cardinalOptimo = cardinalActual;
            }
            cardinalActual-- ; //vuelta atrás, ensaya exclusión de objeto i
      }
         /* proceso de exclusión del objeto i para seguir
                      la búsqueda sistemática con el objeto i+1 */
      valExclusion = va - valorObjs[i-1]; /* decrementa el valor
                                               del objeto excluido */
      if (valExclusion > mejorValor)     /* se puede alcanzar una mejor
                                  selección, sino poda la búsqueda */
         if (i < n)
            seleccionOptima(i+1, pt, valExclusion);
```

```
        else
        {
           mejorValor = valExclusion;
           System.arraycopy(actual,0,optimo,0,cardinalActual);
           cardinalOptimo = cardinalActual;
      }
    }
  }
```

RESUMEN

Un método o función se dice que es recursivo si tiene una o más sentencias que son llamadas a sí mismas. La recursividad puede ser directa o indirecta; ésta ocurre cuando el método `f()` llama a `p()` y ésta, a su vez, llama a `f()`. La recursividad es una alternativa a la iteración en la resolución de algunos problemas matemáticos aunque, en general, es preferible la implementación iterativa debido a que es más eficiente. Los aspectos más importantes a tener en cuenta en el diseño y construcción de métodos recursivos son los siguientes:

- Un algoritmo recursivo contiene dos tipos de casos: uno o más casos que incluyen al menos una llamada recursiva y uno o más casos de terminación o parada del problema, en los que éste se soluciona sin ninguna llamada recursiva, sino con una sentencia simple. De otro modo, un algoritmo recursivo debe tener dos partes: una parte de terminación en la que se deja de hacer llamadas, es el caso base, y una llamada recursiva con sus propios parámetros.
- Muchos problemas tienen naturaleza recursiva y la solución más fácil es mediante un método recursivo. De igual modo, aquellos problemas que no entrañen una solución recursiva se deberán seguir resolviendo mediante algoritmos iterativos.
- Las métodos con llamadas recursivas utilizan memoria extra en las llamadas; existe un límite en las llamadas, que depende de la memoria de la computadora. En caso de superar este límite ocurre un error de *overflow*.
- Cuando se codifica un método recursivo, se debe comprobar siempre que tiene una condición de terminación, es decir, que no se producirá una recursión infinita. Durante el aprendizaje de la recursividad es usual que se produzca ese error.
- Para asegurarse de que el diseño de un método recursivo es correcto, se deben cumplir las siguientes tres condiciones:
  1. No existir recursión infinita. Una llamada recursiva puede conducir a otra llamada recursiva y ésta conducir a otra, y así sucesivamente; cada llamada debe aproximarse más a la condición de terminación.
  2. Para la condición de terminación, el método devuelve el valor correcto para ese caso.
  3. En los casos que implican llamadas recursivas: si cada uno de los métodos devuelve un valor correcto, entonces el valor final devuelto por el método es el valor correcto.
- Una de las técnicas más utilizadas en la resolución de problemas es la denominada *"divide y vence"*. La implementación de estos algoritmos se puede realizar con métodos recursivos.

- Los algoritmos del tipo *vuelta atrás* o *backtracking* se caracterizan por realizar una búsqueda sistemática, exhaustiva, de la solución. Prueba con todas las posibilidades que se tienen para realizar una tarea que vaya encaminada hacia la solución. Cada tarea se expresa por una llamada recursiva, los elementos de la tarea se apuntan (se guardan). En la *vuelta atrás,* retorno de la llamada recursiva, se determina si se alcanzó la solución y en caso contrario, se borra la anotación para probar de nuevo con otra posibilidad.
- La selección óptima se puede considerar como una variante de los algoritmos de *vuelta atrás*, en la que se buscan no sólo los elementos que cumplan una condición, sino que éstos sean los mejores, según el criterio que se haya establecido como *mejor.*

EJERCICIOS

5.1. Convierta el siguiente método iterativo en recursivo. El método calcula un valor aproximado de *e*, la base de los logaritmos naturales, sumando las series

1 + 1/1 ! + 1/2 ! + ... + 1/n !

hasta que los términos adicionales no afecten a la aproximación

```
static public double loge()
{
   double enl, delta, fact;
   int n;
   enl = fact = delta = 1.0;
   n = 1;
   do
   {
      enl += delta;
      n++;
      fact * = n:
      delta = 1.0 / fact;
   } while (enl != enl + delta);
   return enl;
}
```

5.2. Explique por qué el siguiente método puede producir un valor incorrecto cuando se ejecute:

```
static public long factorial (long n)
{
   if (n == 0 || n == 1)
      return 1;
   else
      return n * factorial (--n);
}
```

5.3. ¿Cuál es la secuencia numérica generada por el método recursivo `f()` en el listado siguiente si la llamada es `f(5)`?

```
long f(int n)
{
   if (n == 0 || n == 1)
      return 1;
  else
      return 3 * f(n - 2) + 2 * f(n - 1);
}
```

5.4. Escribir un método recursivo `int vocales(String)` para calcular el número de vocales de una cadena.

5.5. Proporcionar métodos recursivos que representen los siguientes conceptos:

a) El producto de dos números naturales.

b) El conjunto de permutaciones de una lista de números.

5.6. Suponer que la función matemática G está definida recursivamente de la siguiente forma:

$$G(x, y) = \begin{cases} \dfrac{1}{G(x-y+1)} & \text{si } x \le y \\ 2x - y & \text{si } x > y \end{cases}$$

siendo x, y enteros positivos. Encontrar el valor de: (a) G(8, 6); (b) G(100, 10).

5.7. Escribir un método recursivo que calcule la función de Ackermann definida de la siguiente forma:

A(m, n) = n + 1	*si m = 0*
A(m, n) = A (m – 1, 1)	*si m >0, y n = 0*
A(m, n) = A(m – 1, A(m, n – 1))	*si m > 0, y n > 0*

5.8. ¿Cuál es la secuencia numérica generada por el método recursivo siguiente, si la llamada es `f(8)`?

```
long f (int n)
{
   if(n == 0||n ==1)
      return 1;
   else if (n % 2 == 0)
      return 2 + f(n - 1);
   else
      return 3 + f(n - 2);
}
```

5.9. ¿Cuál es la secuencia numérica generada por el método recursivo siguiente?

```
int f(int n)
{
   if (n == 0)
      return 1;
   else if (n == 1)
      return 2;
   else
      return 2*f(n - 2) + f(n - 1);
}
```

5.10. El elemento mayor de un array entero de *n-elementos* se puede calcular recursivamente. Suponiendo que el método:

```
static public int max(int x, int y);
```

devuelve el mayor de dos enteros x e y, definir el método

```
int maxarray(int [] a, int n);
```

que utiliza recursión para devolver el elemento mayor de `a`

Condición de parada: `n == 1`

Incremento recursivo: `maxarray = max(max(a[0]...a[n-2]),a[n-1])`

5.11. Escribir un método recursivo,

```
int product(int[]v, int b);
```

que calcule el producto de los elementos del array `v` mayores que `b`.

5.12. El Ejercicio 5.6 define recursivamente una función matemática. Escribir un método que no utilice la recursividad para encontrar valores de la función.

5.13. La resolución recursiva de las Torres de Hanoi ha sido realizada con dos llamadas recursivas. Volver a escribir la solución con una sola llamada recursiva.

Nota: Sustituir la última llamada por un bucle *repetir-hasta*.

5.14. Aplicar el esquema de los algoritmos *divide y vence* para qué dadas las coordenadas (x,y) de dos puntos en el plano, que representan los extremos de un segmento, se dibuje el segmento.

5.15. Escribir un método recursivo para transformar un número entero en una cadena con el signo y los dígitos de que consta: `String entroaCadena(int n)`.

PROBLEMAS

5.1. La expresión matemática *C(m, n)* en el mundo de la teoría combinatoria de los números, representa el número de combinaciones de *m* elementos tomados de *n* en *n* elementos.

$$C(m, n) = \frac{m!}{n!\,(m-n)!}$$

Escribir una aplicación en la que se dé entrada a los enteros *m, n* y calcule *C(m, n)* donde *n!* es el factorial de *n*.

5.2. Un palíndromo es una palabra que se escribe exactamente igual leída en un sentido o en otro. Palabras tales como *level, deed, ala*, etc. son ejemplos de palíndromos. Aplicar un algoritmo *divide y vence* para determinar si una palabra es palíndromo. Escribir un método recursivo que implemente el algoritmo. Escribir una aplicación en la que se lea una cadena hasta que ésta sea un palíndromo.

5.3. Escribir una aplicación en la que un método recursivo liste todos los subconjuntos de `n` letras para un conjunto dado de `m` letras, por ejemplo para `m = 4` y `n = 2`.

`[A, C, E, K]` → `[A, C], [A, E], [A, K], [C, E,], [C, K], [E, K]`

Nota: el número de subconjuntos es $C_{4,2}$.

5.4. El problema de las ocho reinas se ha resuelto en este capítulo de tal forma que en el momento de encontrar una solución se detiene la búsqueda de más soluciones. Modificar el algoritmo de tal forma que el método recursivo escriba todas las soluciones.

5.5. Escribir una aplicación que tenga como entrada una secuencia de números enteros positivos (mediante una variable entera). El programa debe hallar la suma de los dígitos de cada entero y encontrar cual es el entero cuya suma de dígitos es mayor. La suma de dígitos ha de hacerse con un método recursivo.

5.6. Desarrollar un método recursivo que cuente el número de números binarios de *n-dígitos* que no tengan dos 1 en una fila. *Sugerencia:* El número comienza con un 0 o un 1. Si comienza con 0, el número de posibilidades se determina por los restantes *n-1* dígitos. Si comienza con 1, ¿cuál debe ser el siguiente?

5.7 Desarrolar una aplicación que lea un número entero positivo $n < 10$ y calcule el desarrollo del polinomio $(x + 1)^n$. Imprimir cada potencia x^i en la forma `x**i`.

Sugerencia:

$$(x + 1)^n = C_{n,n}x^n + C_{n,\,n-1}x^{n-1} + C_{n,\,n-2}x^{n-2} + ... +C_{n,\,2}x^2 + C_{n,1}x^1 + C_{n,0}x^0$$

donde $C_{n,n}$ y $C_{n,0}$ son 1 para cualquier valor de *n.*

La relación de recurrencia de los coeficientes binomiales es:

C(n, 0) = 1

C(n, n) = 1

C(n, k) = C(n-1, k-1) + C(n-1, k)

Estos coeficientes constituyen el famoso Triángulo de Pascal y será preciso definir el método que genera el triángulo

```
        1
      1   1
    1   2   1
  1   3   3   1
1   4   6   4   1
...
```

5.8. Sea `A` una matriz cuadrada de `n x n` elementos, el determinante de `A` se puede definir de manera recursiva:

a) Si `n = 1`, entonces `Deter(A)=` $a_{1,1}$.

b) Para `n > 1`, el determinante es la suma alternada de productos de los elementos de una fila o columna elegida al azar por sus menores complementarios. A su vez, los menores complementarios son los determinantes de orden `n-1` obtenidos al suprimir la fila y la columna en que se encuentra el elemento.

Puede expresarse:

$$Det(A) = \sum_{i=1}^{n} (-1)^{i+j} * A[i,j] * Det(Menor(A[i,j])); \text{ para cualquier columna } j$$

o

$$Det(A) = \sum_{j=1}^{n} (-1)^{i+j} * A[i,j] * Det(Menor(A[i,j])); \text{ para cualquier columna } i$$

Se observa que la resolución del problema sigue la estrategia de los algoritmos *divide y vence*.

Escribir una aplicación que tenga como entrada los elementos de la matriz `A` y tenga como salida la matriz `A` y el determinante de `A`. Elegir la fila `1` para calcular el determinante.

5.9. Escribir una aplicación que transforme números enteros en base 10 a otro en base b, siendo ésta de 8 a 16. La transformación se ha de realizar siguiendo una estrategia recursiva.

5.10. Escribir una aplicación para resolver el problema de la subsecuencia creciente más larga. La entrada es una secuencia de n números $a_1, a_2, a_3, \ldots, a_n$; hay que encontrar la subsecuencia mas larga $a_{i1}, a_{i2}, \ldots a_{ik}$ tal que $a_{i1} < a_{i2} < a_{i3} \ldots < a_{ik}$ y que $i1 < i2 < i3 < \ldots < ik$. El programa escribirá dicha subsecuencia.
Por ejemplo, si la entrada es `3,2,7,4,5,9,6,8,1`, la subsecuencia creciente más larga tiene longitud cinco: `2,4,5,6,8`.

5.11. El sistema monetario consta de monedas de valores $p_1, p_2, p_3, \ldots, p_n$ (orden creciente). Escribir una aplicación que tenga como entrada el valor de las `n` monedas, en orden creciente, y una cantidad `x` de cambio. Calcule:

a) El número mínimo de monedas que se necesitan para dar el cambio `x`.

b) El número de formas diferentes de dar el cambio de la cantidad `x` con la p_i monedas.

Aplicar técnicas recursivas para resolver el problema.

5.12. En un tablero de ajedrez, se coloca un alfil en la posición (x_0, y_0) y un peón en la posición `(1,j)`, siendo `1 <= j <= 8`. Se pretende encontrar una ruta para el peón que llegue a la fila 8 sin ser comido por el alfil, siendo el único movimiento permitido para el peón el de avance desde la posición `(i,j)` a la posición `(i+1,j)`. Si se encuentra que el peón está amenazado por el alfil en la posición `(i,j)`, entonces debe retroceder a la fila `1`, columna `j+1` o `j-1` `{(1,j+1), (1,j-1)}`.
Escribir una aplicación para resolver el supuesto problema. Hay que tener en cuenta que el alfil ataca por diagonales.

5.13. Dados `n` números enteros positivos, encontrar una combinación de ellos que mediante sumas o restas totalicen exactamente un valor objetivo `Z`. La aplicación debe tener como entrada los `n` números y el objetivo `Z`; la salida ha de ser la combinación de números con el operador que le corresponda.

Tener en cuenta que pueden formar parte de la combinación los `n` números o parte de ellos.

5.14. Dados n números, encontrar la combinación con sumas o restas que más se aproxime a un objetivo Z. La aproximación puede ser por defecto o por exceso. La entrada son los n números, y el objetivo y la salida son la combinación más proxima al objetivo.

5.15. Podemos emular un laberinto con una matriz n x n en la que los pasos libres estén representados por un carácter (el blanco por ejemplo) y los muros por otro carácter (el □ por ejemplo). Escribir una aplicación que se genere aleatoriamente un laberinto, se pidan las coordenadas de entrada (la fila será la 1), las coordenadas de salida (la fila será la n) y encontrar todas las rutas que nos lleven de la entrada a la salida.

5.16. Realizar las modificaciones necesarias en el problema anterior para encontrar la ruta más corta, considerando ésta la que pasa por un menor número de casillas.

5.17. Una región castellana está formada por n pueblos dispersos. Hay conexiones directas entre algunos de ellos, y entre otros no existe conexión, aunque puede haber un camino. Escribir una aplicación que tenga como entrada la matriz que representa las conexiones directas entre pueblos, de tal forma que el elemento M(i,j) de la matriz sea:

$$M(i,j) = \begin{cases} 0 & \text{si no hay conexión directa entre pueblo } i \text{ y pueblo } j. \\ d & \text{hay conexión entre pueblo } i \text{ y pueblo } j \text{ de distancia } d. \end{cases}$$

Tenga también como entrada un par de pueblos (x,y). La aplicación tiene que encontrar un camino entre ambos pueblos utilizando técnicas recursivas. La salida ha de ser la ruta que se ha de seguir para ir de x a y junto a la distancia de la ruta.

5.18. En el programa escrito en el ejercicio anterior, hacer las modificaciones necesarias para encontrar todos los caminos posibles entre el par de pueblos (x,y).

5.19. Un número entero sin signo, m, se dice que es *dos_tres_cinco* si cumple las características:

- Todos los dígitos de m son distintos.
- La suma de los dígitos que ocupan posiciones pares es igual a la suma de los dígitos que ocupan posiciones múltiplos de tres más la suma de los dígitos que ocupan posiciones múltiplos de cinco.

Implementar una aplicación que genere todos los números enteros de cinco o más cifras que séan *dos_tres_cinco.*

CAPITULO 6

Algoritmos de ordenación y búsqueda

Objetivos

Una vez que se haya leído y estudiado este capítulo, usted podrá:

- Conocer los algoritmos basados en el intercambio de elementos.
- Conocer el algoritmo de ordenación por inserción.
- Conocer el algoritmo de selección.
- Distinguir entre los algoritmos de ordenación basados en el intercambio y en la inserción.
- Saber la eficiencia de los métodos básicos de ordenación.
- Conocer los métodos más eficientes de ordenación.
- Aplicar métodos mas eficientes de ordenación de arrays.
- Ordenar vectores de objetos.
- Diferenciar entre búsqueda secuencial y búsqueda binaria.

Contenido

Conceptos clave

- ♦ Búsqueda en listas: búsqueda secuencial y búsqueda binaria.
- ♦ Complejidad cuadrática.
- ♦ Complejidad logarítmica.
- ♦ Ordenación alfabética.
- ♦ Ordenación numérica.
- ♦ Ordenación por burbuja.
- ♦ Ordenación por inserción.
- ♦ Ordenación por intercambio.
- ♦ Ordenación por selección.
- ♦ Ordenación rápida.
- ♦ Residuos.
- ♦ Vector de objetos.

Para profundizar (página web: www.mhe.es/joyanes)

- ▪ Ordenación por *burbuja*.
- ▪ Ordenación *Binsort* y *Radixsort*.

INTRODUCCIÓN

Muchas actividades humanas requieren que diferentes colecciones de elementos utilizados se pongan en un orden específico. Las oficinas de correo y las empresas de mensajería ordenan el correo y los paquetes por códigos postales con el objeto de conseguir una entrega eficiente; las facturas telefónicas se ordenan por la fecha de las llamadas; los anuarios o listines telefónicos se ordenan por orden alfabético de apellidos con el fin último de encontrar fácilmente el número de teléfono deseado; los estudiantes de una clase en la universidad se ordenan por sus apellidos o por los números de expediente. Por estas circunstancias una de las tareas que realizan más frecuentemente las computadoras en el procesamiento de datos es la *ordenación*.

El estudio de diferentes métodos de ordenación es una tarea intrínsecamente interesante desde un punto de vista teórico y, naturalmente, práctico. Este capítulo estudia los algoritmos y las técnicas de ordenación más usuales y su implementación en Java; también la manera de ordenar objetos con la funcionalidad que proporcionan las clases en Java. De igual modo, se estudiará el análisis de los diferentes métodos de ordenación con el objetivo de conseguir la máxima eficiencia en su uso real.

En el capítulo se analizarán los métodos básicos y los más avanzados empleados en programas profesionales.

6.1. ORDENACIÓN

La **ordenación** o **clasificación** de datos (*sort,* en inglés) es una operación consistente en disponer un conjunto —estructura— de datos en algún determinado orden con respecto a uno de los *campos* de los elementos del conjunto. Por ejemplo, cada elemento del conjunto de datos de una guía telefónica tiene un campo nombre, un campo dirección y un campo número de teléfono; la guía telefónica está dispuesta en orden alfabético de nombres. Los elementos numéricos se pueden ordenar en orden creciente o decreciente de acuerdo al valor numérico del elemento. En terminología de ordenación, el elemento por el cual está ordenado un conjunto de datos (o se está buscando) se denomina *clave*.

Una colección de datos *(estructura)* puede ser almacenada en memoria central o en archivos de datos externos guardados en unidades de almacenamiento magnético (discos, cintas, CD-ROM, DVD, etc.). Cuando los datos se guardan en un *array,* en una *lista enlazada* o en un *árbol,* se denomina *ordenación interna;* estos datos se almacenan exclusivamente para tratamientos internos que se utilizan para gestión masiva de datos, se guardan en *arrays* de una o varias dimensiones. Si los datos están almacenados en un archivo, el proceso de ordenación se llama *ordenación externa*.

A recordar

Existen dos técnicas de ordenación fundamentales en gestión de datos: *ordenación de listas* y *ordenación de archivos*. Los métodos de ordenación se conocen como *internos* o *externos* según los elementos a ordenar estén en la memoria principal o en la memoria externa.

Este capítulo estudia los métodos de ordenación de datos que están en la memoria principal, *ordenación interna*.

Una *lista está ordenada por la clave k* si la lista está en orden ascendente o descendente con respecto a esa clave. La lista está en *orden ascendente* si:

```
i < j    implica que    k[i] <= k[j]
```

y está en *orden descendente* si:

```
i > j    implica que    k[i] <= k[j]
```

para todos los elementos de la lista. Por ejemplo, para una guía telefónica, la lista está clasificada en orden ascendente por el campo clave *k,* donde *k*[*i*] es el nombre del abonado (apellidos, nombre).

```
4     5     14    21    32    45                        orden ascendente
75    70    35    16    14    12                        orden descendente
Zacarias Rodriguez Martinez Lopez Garcia                orden descendente
```

Los métodos (algoritmos) de ordenación son numerosos, por ello se debe prestar especial atención en su elección. ¿Cómo se sabe cuál es el mejor algoritmo? La *eficiencia* es el factor que mide la calidad y el rendimiento de un algoritmo. En el caso de la operación de ordenación, se suelen seguir dos criterios a la hora de decidir qué algoritmo —de entre los que resuelven la ordenación— es el más eficiente: 1) *tiempo menor de ejecución en computadora*; 2) *menor número de instrucciones*. Sin embargo, no siempre es fácil efectuar estas medidas: puede no disponerse de instrucciones para medida del tiempo —aunque no sea éste el caso del lenguaje Java—, y las instrucciones pueden variar, dependiendo del lenguaje y del propio estilo del programador. Por esta razón, el mejor criterio para medir la eficiencia de un algoritmo es aislar una operación específica, clave en la ordenación y contar el número de veces que se realiza. Así, en el caso de los algoritmos de ordenación, se utilizará como medida de su eficiencia el número de comparaciones entre elementos efectuados. El algoritmo de *ordenación A* será más eficiente que el *B* si requiere menor número de comparaciones. En la ordenación de los elementos de un vector, el número de comparaciones será *función* del número de elementos (*n*) del vector *(array)*. Por consiguiente, se puede expresar el número de comparaciones en términos de *n* (por ejemplo, $n+4$ o bien n^2) en lugar de números enteros (por ejemplo, *325*).

Todos los métodos de este capítulo, normalmente —para comodidad del lector— se *ordenan de modo ascendente* sobre vectores o listas *(arrays* unidimensionales).

Los métodos de ordenación se suelen dividir en dos grandes grupos:

- *directos* — burbuja, selección, inserción
- *indirectos* (avanzados) — *shell*, ordenación rápida, ordenación por mezcla, *radixsort*

En el caso de listas pequeñas, los métodos directos se muestran eficientes, sobre todo, porque los algoritmos son sencillos, por lo que su uso es muy frecuente. Sin embargo, en listas grandes, estos métodos se muestran ineficaces y es preciso recurrir a los métodos avanzados.

6.2. ALGORITMOS DE ORDENACIÓN BÁSICOS

Existen diferentes algoritmos de ordenación elementales o básicos cuyos detalles de implementación se pueden encontrar en diferentes libros de algoritmos. La enciclopedia de referencia es [KNUTH 1973][1] y, sobre todo, la 2ª edición publicada en el año 1998 [KNUTH 1998][2]. Los algoritmos presentan diferencias entre ellos que los convierten en más o menos eficientes y

---

1 [KNUTH 1973] Donald E. Knuth. *The Art of Computer Programming*. *Sorting and Searching*. Volumen 3 Addison-Wesley, 1973.

2 [KNUTH 1998] Donald E. Knuth. *The Art of Computer Programming. Sorting and Searching.* Volumen 3. Second Edition. Addison-Wesley, 1998.

prácticos según sea la rapidez y eficiencia demostrada por cada uno de ellos. Los algoritmos básicos de ordenación más simples y clásicos son:

- Ordenación por selección.
- Ordenación por inserción.
- Ordenación por burbuja.

Los métodos más recomendados son el de *selección* y el de *inserción,* aunque se estudiará el método de *burbuja* por ser el más sencillo, pero también es el más *ineficiente;* por esta causa no se recomienda su uso, pero si conocer su técnica (véase página web).

Las técnicas que se estudian a continuación considerarán, esencialmente, la ordenación de elementos de una lista *(array)* en orden ascendente. En cada caso se analiza la eficiencia computacional del algoritmo.

Con el objeto de facilitar el aprendizaje al lector, y aunque no sea un método utilizado por su poca eficiencia se describe en primer lugar el método de *ordenación por intercambio,* debido a la sencillez de su técnica y con la finalidad de que el lector no introducido en los algoritmos de ordenación pueda comprender su funcionamiento y luego asimile más eficazmente los tres algoritmos básicos ya citados y los avanzados, que se estudian más adelante.

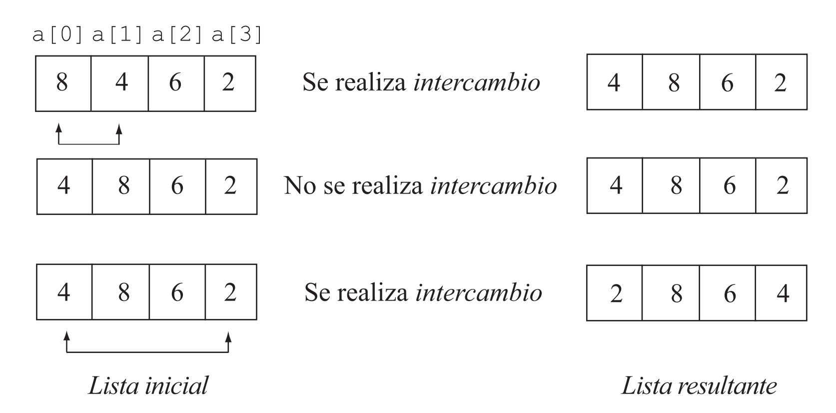
6.3. ORDENACIÓN POR INTERCAMBIO

El algoritmo de ordenación tal vez más sencillo sea el denominado de *intercambio,* que ordena los elementos de una lista en orden ascendente. El algoritmo se basa en la lectura sucesiva de la lista a ordenar, comparando el elemento inferior de la lista con los restantes y efectuando un intercambio de posiciones cuando el orden resultante de la comparación no sea el correcto.

El algoritmo se ilustra con la lista original `8, 4, 6, 2` que ha de convertirse en la lista ordenada `2, 4, 6, 8`. El algoritmo efectúa *n-1* pasadas (3 en el ejemplo), siendo *n* el número de elementos, realizando las comparaciones indicadas en las figuras.

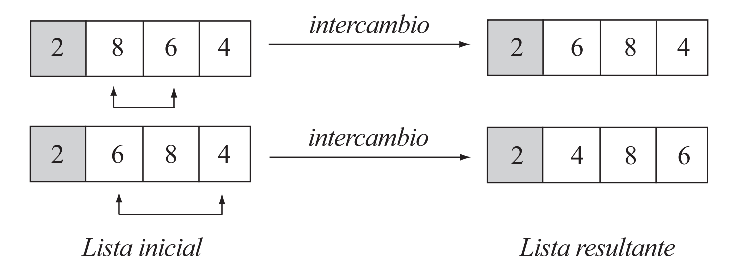
Pasada 1

El elemento de índice `0 (a[0])` se compara con cada elemento posterior de la lista de índices `1, 2` y `3`. En cada comparación se comprueba si el elemento siguiente es más pequeño que el elemento de índice `0` y en ese caso, se intercambian. Después de terminar todas las comparaciones, el elemento más pequeño se sitúa en el índice 0.



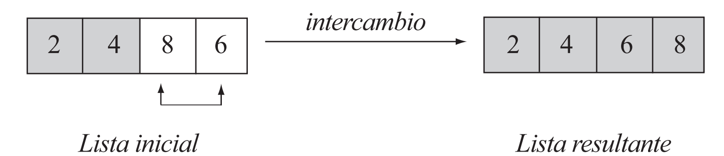
Pasada 2

El elemento más pequeño ya está en la posición de índice `0`, y se considera la sublista restante `8, 6, 4`. El algoritmo continúa comparando el elemento de índice `1` con los elementos posteriores de índices `2` y `3`. Por cada comparación, si el elemento mayor está en el índice `1` se intercambian los elementos. Después de hacer todas las comparaciones, el segundo elemento más pequeño de la lista se almacena en el índice `1`.



Pasada 3

La sublista a considerar ahora es `8`, `6`, ya que `2`, `4` está ordenada. Se produce entre los dos elementos de la sublista una comparación única.



6.3.1. Codificación del algoritmo de ordenación por intercambio

El método `ordIntercambio()` implementa el algoritmo descrito utilizando dos bucles anidados. El tipo de los elementos de la lista es entero, es válido cualquier otro tipo ordinal `(double, char...)`. Suponiendo que la lista es de tamaño *n*, el rango del bucle externo va desde el índice `0` hasta `n-2`. Por cada índice `i`, se comparan los elementos posteriores de índices $j = i + 1,\ i + 2, ..., n-1$. El intercambio *(swap)* de los elementos `a[i],a[j]` se realiza en el método `intercambiar()`:

```
public static void intercambiar(int []a, int i, int j)
{
   int aux = a[i];
   a[i] = a[j];
   a[j]= aux ;
}
```

El *array* (arreglo) dispone de tantos elementos como posiciones creadas. Por ello el método `ordIntercambio()` tiene un único argumento `(int [] a)`; el atributo `a.length` es el valor del número de elementos `(n)`.

```
public static void ordIntercambio (int a[])
```

```
{
   int i, j;
   for (i = 0 ; i < a.length-1; i++)
      // sitúa mínimo de a[i+1]...a[n-1] en a[i]
      for (j = i+1 ; j < a.length; j++)
         if (a[i] > a[j])
         {
            intercambiar(a, i, j);
         }
}
```

6.3.2. Complejidad del algoritmo de ordenación por intercambio

El algoritmo consta de dos bucles anidados, está *dominado* por los dos bucles, de ahí que el análisis del algoritmo en relación a la complejidad sea inmediato; siendo `n` el número de elementos, el primer bucle hace `n-1` pasadas y el segundo `n-i-1` comparaciones en cada pasada (`i` es el índice del bucle externo, `i = 0 .. n-2`). El número total de comparaciones se obtiene desarrollando la sucesión matemática formada para los distintos valores de `i`:

```
n-1, n-2, n-3, ... , 1
```

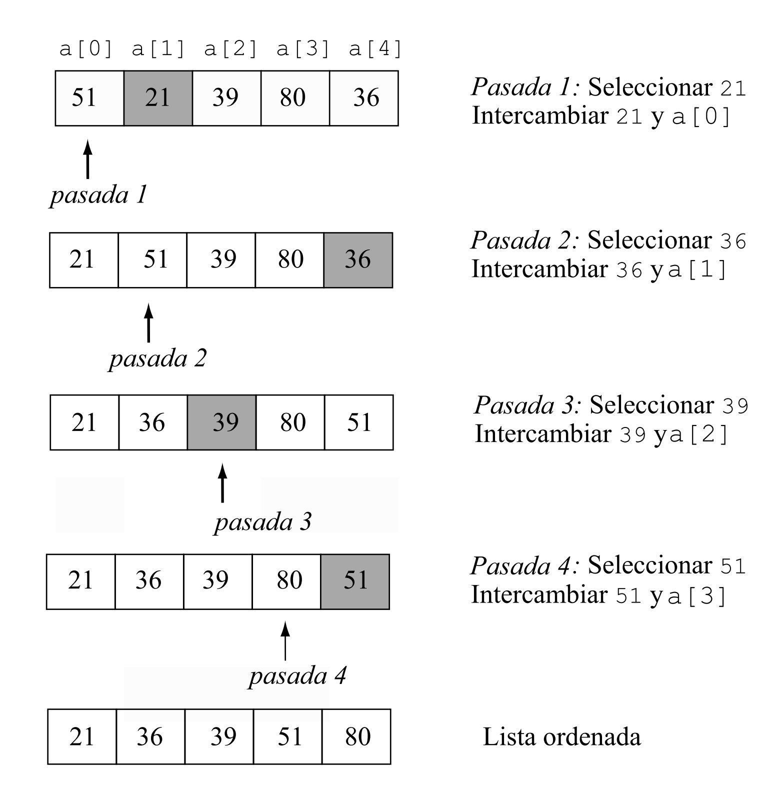
La suma de los términos de la sucesión. La ordenación completa requiere $\frac{(n-1)(n-1)}{2}$ comparaciones y un número similar de intercambios *en el peor de los casos*. Entonces, el número de comparaciones es $(n-1) * (n-1)/2 = (n^2 - 2n + 1)/2$ y *en el peor de los casos,* los mismos intercambios. El término dominante es n^2, por tanto, la complejidad es $0(n^2)$.

6.4. ORDENACIÓN POR SELECCIÓN

Considérese el algoritmo para ordenar un *array* `a[]` de enteros en orden ascendente, es decir, si el *array* `a[]` tiene *n* elementos, se trata de ordenar los valores del *array* de modo que `a[0]` sea el valor más pequeño, el valor almacenado en `a[1]` sea el siguiente más pequeño, y así hasta `a[n-1]` que ha de contener el elemento de mayor valor. El algoritmo se apoya en las *pasadas* que intercambian el elemento más pequeño, sucesivamente con el elemento de la lista, `a[]`, que ocupa la posición igual al orden de *pasada* (hay que considerar el índice `0`). La *pasada* inicial busca el elemento más pequeño de la lista y se intercambia con `a[0]`, primer elemento de la lista.

Después de terminar esta primera pasada, el frente de la lista está ordenado y el resto de la lista `a[1], a[2]...a[n-1]` permanece desordenado. La siguiente *pasada* busca en esta lista desordenada y *selecciona* el elemento más pequeño, que se almacena entonces en la posición `a[1]`. De este modo, los elementos `a[0]` y `a[1]` están ordenados y la sublista `a[2], a[3]... a[n-1]` desordenado; entonces, se selecciona el elemento más pequeño y se intercambia con `a[2]`. El proceso continúa hasta realizar `n-1` *pasadas,* en ese momento la lista desordenada se reduce a un elemento (el mayor de la lista) y el *array* completo queda ordenado.

Un ejemplo práctico ayudará a la comprensión del algoritmo. Consideremos un *array* `a[]` con 5 valores enteros `51, 21, 39, 80, 36`:



6.4.1. Codificación del algoritmo de selección

El método `ordSeleccion()` ordena un *array* de números reales de `n` elementos, `n` coincide con el atributo `length` del *array*. En la pasada `i`, el proceso de selección explora la sublista `a[i]` a `a[n-1]` y fija el índice del elemento más pequeño. Después de terminar la exploración, los elementos `a[i]` y `a[indiceMenor]` se intercambian; operación que se realiza llamando al método `intercambiar()` escrito en el apartado 6.3 (es necesario cambiar tipo `int` por tipo `double`).

```
    /*
       ordenar un array de n elementos de tipo double
       utilizando el algoritmo de ordenación por selección
    */

  public static void ordSeleccion (double a[])
  {
    int indiceMenor, i, j, n;
    n = a.length;
    for (i = 0; i < n-1; i++)
    {
      // comienzo de la exploración en índice i
      indiceMenor = i;
      // j explora la sublista a[i+1]..a[n-1]
      for (j = i+1; j < n; j++)
        if (a[j] < a[indiceMenor])
          indiceMenor = j;
```

```
         // sitúa el elemento mas pequeño en a[i]
         if (i != indiceMenor)
            intercambiar(a, i, indiceMenor);
      }
   }
```

6.4.2. Complejidad del algoritmo de selección

El análisis del algoritmo, con el fin de determinar la función *tiempo de ejecución* `t(n)`, es sencillo y claro, ya que requiere un número fijo de comparaciones que sólo dependen del tamaño del *array* y no de la distribución inicial de los datos. El término *dominante* del algoritmo es el bucle externo que anida a un bucle interno. Por ello, el número de comparaciones que realiza el algoritmo es el número decreciente de iteraciones del bucle interno: `n-1, n-2, ... 2, 1` (`n` es el número de elementos). La suma de los términos de la sucesión se ha obtenido en el Apartado 6.3.2, y se ha comprobado que depende de n^2. Como conclusión, la complejidad del algoritmo de selección es $0(n^2)$.

6.5. ORDENACIÓN POR INSERCIÓN

El método de ordenación por inserción es similar al proceso típico de ordenar tarjetas de nombres (cartas de una baraja) por orden alfabético consistente en insertar un nombre en su posición correcta dentro de una lista que ya está ordenada. El proceso en el caso de la lista de enteros es:

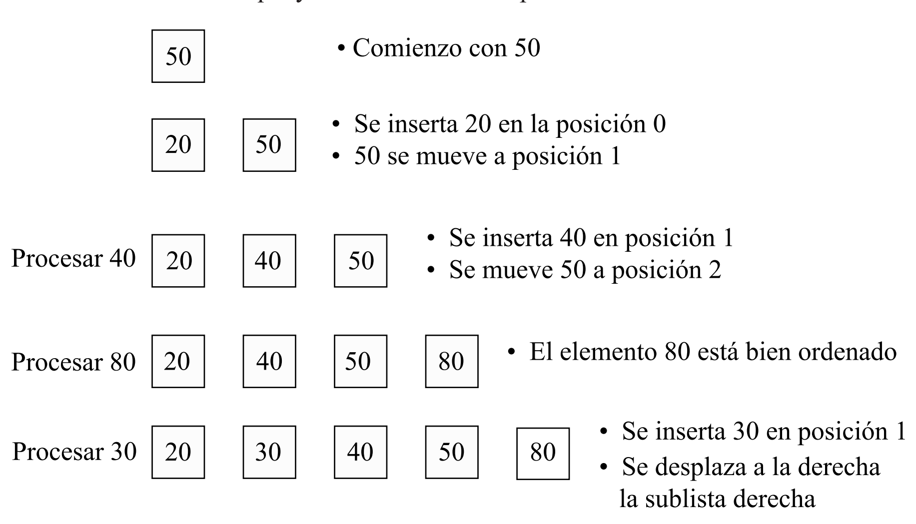


Figura 6.1 Método de ordenación por inserción

6.5.1. Algoritmo de ordenación por inserción

El algoritmo correspondiente a la ordenación por inserción contempla los siguientes pasos:

1. El primer elemento `a[0]` se considera ordenado; es decir, la lista inicial consta de un elemento.
2. Se inserta `a[1]` en la posición correcta; delante o detrás de `a[0]`, dependiendo de si es menor o mayor.

3. Por cada bucle o iteración `i` *(desde* `i=1` *hasta* `n-1`) se explora la sublista `a[i-1] ... a[0]` buscando la posición correcta de inserción de `a[i]`; a la vez, se mueven hacia abajo (a la derecha en la sublista) una posición todos los elementos mayores que el elemento a insertar `a[i]`, para dejar vacía esa posición.
4. Insertar el elemento `a[i]` a la posición correcta.

6.5.2. Codificación del algoritmo de ordenación por inserción

La codificación del algoritmo se realiza en el método `ordInsercion()`. Se pasa como argumento el *array,* `a[]`, que se va a ordenar de modo creciente; el número de elementos a ordenar coincide con el atributo del *array* `length`. Los elementos del *array* son de tipo entero; en realidad, pueden ser de cualquier tipo básico y ordinal de Java.

```
public static void ordInsercion (int [] a)
{
   int i, j;
   int aux;
   for (i = 1; i < a.length; i++)
   {
      /*  indice j es para explorar la sublista a[i-1]..a[0] buscando la
          posicion correcta del elemento destino*/
      j = i;
      aux = a[i];
      // se localiza el punto de inserción explorando hacia abajo
      while (j > 0 && aux < a[j-1])
      {
         // desplazar elementos hacia arriba para hacer espacio
         a[j] = a[j-1];
         j--;
      }
      a[j] = aux;
   }
}
```

6.5.3. Complejidad del algoritmo de inserción

A la hora de analizar este algoritmo, se observa que el número de instrucciones que realiza depende del bucle automático `for` (bucle externo) que anida al bucle condicional `while`. Siendo `n` el número de elementos (`n == a.length`), el bucle externo realiza `n-1` *pasadas;* por cada una de ellas y en *el peor de los casos* (`aux` siempre menor que `a[j-1]`), el bucle interno `while` itera un número creciente de veces que da lugar a la sucesión `1, 2, 3, ... n-1` (para `i == n-1`). La suma de los términos de la sucesión se ha obtenido en el apartado 6.3.2, y se ha comprobado que el término dominante es n^2. Como conclusión, la complejidad del algoritmo de inserción es $0(n^2)$.

6.6. ORDENACIÓN *SHELL*

La ordenación Shell debe el nombre a su inventor, D. L. Shell. Se suele denominar también *ordenación por inserción con incrementos decrecientes*. Se considera que el método Shell es una mejora del método de inserción directa.

En el algoritmo de inserción, cada elemento se compara con los elementos contiguos de su izquierda, uno tras otro. Si el elemento a insertar es el mas pequeño, hay que realizar muchas comparaciones antes de colocarlo en su lugar definitivo.

El algoritmo de Shell modifica los saltos contiguos resultantes de las comparaciones por saltos de mayor tamaño, y con ello se consigue que la ordenación sea más rápida. Generalmente, se toma como salto inicial *n/2* (siendo *n* el número de elementos), y luego se reduce el salto a la mitad en cada repetición hasta que sea de tamaño 1. El Ejemplo 6.1 ordena una lista de elementos siguiendo paso a paso el método de Shell.

Ejemplo 6.1

Obtener las secuencias parciales del vector al aplicar el método Shell para ordenar de modo creciente la lista:

6 5 2 3 4 0

El número de elementos que tiene la lista es 6, por lo que el salto inicial es *6/2 = 3*. La siguiente tabla muestra el número de recorridos realizados en la lista con los saltos correspondiente.

Recorrido	*Salto*	*Intercambios*	*Lista*
1	3	(6,2),(5,4),(6,0)	2 4 0 3 5 6
2	3	(2,0)	0 4 2 3 5 6
3	3	ninguno	0 4 2 3 5 6
salto 3/2=1			
4	3	(4,2),(4,3)	0 2 3 4 5 6
5	1	ninguno	0 2 3 4 5 6

6.6.1. Algoritmo de ordenación Shell

Los pasos a seguir por el algoritmo para una lista de `n` elementos son:

1. Se divide la lista original en `n/2` grupos de dos, considerando un incremento o salto entre los elementos de `n/2`.
2. Se clasifica cada grupo por separado, comparando las parejas de elementos, y si no están ordenados se intercambian.
3. Se divide ahora la lista en la mitad de grupos `(n/4)`, con un incremento o salto entre los elementos también mitad `(n/4)`, y nuevamente se clasifica cada grupo por separado.
4. Así sucesivamente, se sigue dividiendo la lista en la mitad de grupos que en el recorrido anterior con un incremento o salto decreciente en la mitad que el salto anterior, y luego clasificando cada grupo por separado.
5. El algoritmo termina cuando se llega a que el tamaño del salto es 1.

Por consiguiente, los recorridos por la lista están condicionados por el bucle,

```
intervalo ← n / 2
mientras (intervalo > 0) hacer
```

Para dividir la lista en grupos y clasificar cada grupo se anida este código,

```
desde i ← (intervalo + 1) hasta n hacer
   j ← i - intervalo
   mientras (j > 0) hacer
      k ← j + intervalo
      si (a[j] <= a[k]) entonces
         j ← 0
      sino
         Intercambio (a[j], a[k]);
         j ← j - intervalo
      fin _ si
   fin _ mientras
fin _ desde
```

donde se observa que se comparan pares de elementos de índice `j` y `k`, `a[j], a[k])`, separados por un *salto* de `intervalo`. Así, si `n = 8`, el primer valor de `intervalo = 4`, y los índices `i = 5`, `j = 1`, `k = 6`. Los siguiente valores que toman `i = 6`, `j = 2`, `k = 7` y así hasta recorrer la lista.

Para realizar un nuevo recorrido de la lista con la mitad de grupos, el intervalo se reduce a la mitad.

```
intervalo ← intervalo / 2
```

Y así se repiten los recorridos por la lista, *mientras intervalo > 0.*

6.6.2. Codificación del algoritmo de ordenación Shell

Al codificar el algoritmo es preciso considerar que Java toma como base en la indexación de *arrays* índice `0` y, por consiguiente, se han de desplazar una posición *a la izquierda* las variables índice respecto a lo expuesto en el algoritmo.

```
public static void ordenacionShell(double a[])
{
   int intervalo, i, j, k;
   int n= a.length;

   intervalo = n / 2;
   while (intervalo > 0)
   {
      for (i = intervalo; i < n; i++)
      {
         j = i - intervalo;
         while (j >= 0)
         {
            k = j + intervalo;
            if (a[j] <= a[k])
               j = -1; // par de elementos ordenado
            else
            {
               intercambiar(a, j, j+1);
               j -= intervalo;
            }
         }
      }
      intervalo = intervalo / 2;
   }
}
```

6.6.3. Análisis del algoritmo de ordenación Shell

A pesar de que el algoritmo tiene tres bucles anidados `(while-for-while)` es más eficiente que el algoritmo de inserción y que cualquiera de los algoritmos simples analizados en los apartados anteriores. El análisis del tiempo de ejecución del algoritmo Shell no es sencillo. Su inventor, Shell, recomienda que el intervalo inicial sea `n/2` y continuar dividiendo el intervalo por la mitad hasta conseguir un intervalo `1` (así se hace en el algoritmo y en la codificación expuestos). Con esta elección se puede probar que el tiempo de ejecución es $0(n^2)$ *en el peor de los casos*, y el tiempo medio de ejecución es $0(n^{3/2})$.

Posteriormente, se han encontrado secuencias de intervalos que mejoran el rendimiento del algoritmo. Un ejemplo de ello consiste en dividir el intervalo por `2.2` en lugar de por la mitad. Con esta nueva secuencia de intervalos se consigue un tiempo medio de ejecución de complejidad menor de $0(n^{5/4})$.

Nota de programación

La codificación del algoritmo Shell con la mejora de hacer el intervalo igual al intervalo anterior dividido por `2.2` puede hacer el intervalo igual a `0`. Si esto ocurre, se ha de codificar que el intervalo sea igual a `1`, en caso contrario, no funcionará el algoritmo.

```
intervalo = (int) intervalo / 2.2;
intervalo = (intervalo == 0) ? 1 : intervalo;
```

6.7. ORDENACIÓN RÁPIDA *(QUICKSORT)*

El algoritmo conocido como *quicksort* (ordenación rápida) recibe su nombre de su autor, Tony Hoare. La idea del algoritmo es simple, se basa en la división en particiones de la lista a ordenar, por ello se puede considerar que aplica la técnica "*divide y vencerás*". El método es, posiblemente, el más pequeño de código, más rápido, más elegante y más interesante y eficiente de los algoritmos conocidos de ordenación.

Este método se basa en dividir los n elementos de la lista a ordenar en dos partes o particiones separadas por un elemento: una partición *izquierda,* un elemento *central* denominado *pivote* o elemento de partición y una partición *derecha*. La partición o división se hace de tal forma que todos los elementos de la primera sublista (partición izquierda) sean menores que todos los elementos de la segunda sublista (partición derecha). Las dos sublistas se ordenan entonces independientemente.

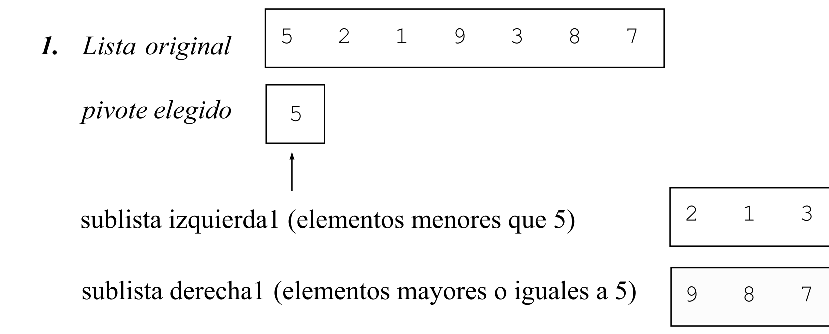
Para dividir la lista en particiones (sublistas) se elige uno de los elementos de la lista y se utiliza como *pivote* o *elemento de partición*. Si se elige una lista cualquiera con los elementos en orden aleatorio, se puede elegir cualquier elemento de la lista como pivote, por ejemplo, el primer elemento de la lista. Si la lista tiene algún orden parcial que se conoce, se puede tomar otra decisión para escogerlo. Idealmente, el pivote se debe elegir de modo que se divida la lista exactamente por la mitad de acuerdo al tamaño relativo de las claves. Por ejemplo, si se tiene una lista de enteros de 1 a 10, 5 o 6 serían pivotes ideales, mientras que 1 o 10 serían elecciones "pobres" de pivotes.

Una vez que el pivote ha sido elegido, se utiliza para ordenar el resto de la lista en dos sublistas: una tiene todas las claves menores que el pivote y la otra, todos los elementos (claves)

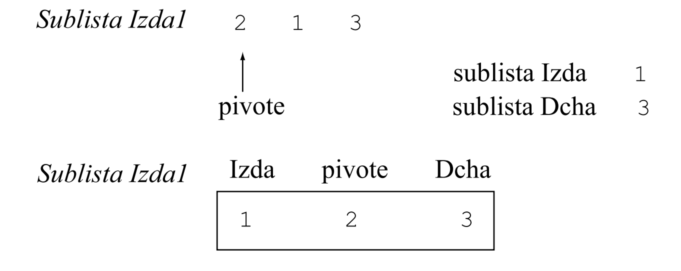
mayores o iguales que el pivote (o al revés). Estas dos listas parciales se ordenan recursivamente utilizando el mismo algoritmo; es decir, se llama sucesivamente al propio algoritmo *quicksort*. La lista final ordenada se consigue concatenando la primera sublista, el pivote y la segunda lista, en ese orden, en una única lista. La primera etapa de *quicksort* es la división o "particionado" recursivo de la lista hasta que todas las sublistas constan de sólo un elemento.

Ejemplo 6.2

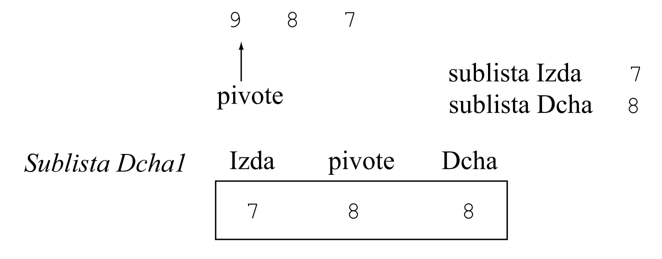
Se ordena una lista de números enteros aplicando el algoritmo quicksort*, como pivote se elige el primer elemento de la lista.*



2. El algoritmo se aplica a la sublista izquierda



3. El algoritmo se aplica a la sublista derecha



4. *Lista ordenada final*

Sublista izquierda	*pivote*	*Sublista derecha*
1 2 3	5	7 8 9

> **A recordar**
>
> El algoritmo *quicksort* requiere una estrategia de partición y la selección idónea del pivote. Las etapas fundamentales del algoritmo dependen del pivote elegido, aunque la estrategia de partición suele ser similar.

6.7.1. Algoritmo *Quicksort*

La primera etapa en el algoritmo de partición es obtener el elemento pivote; una vez que se ha seleccionado, se ha de buscar el sistema para situar en la sublista izquierda todos los elementos menores que el pivote y en la sublista derecha todos los elementos mayores que él. Supongamos que todos los elementos de la lista son distintos, aunque será preciso tener en cuenta los casos en que existan elementos idénticos. En el Ejemplo 6.3 se elige como pivote el elemento central de la lista actual.

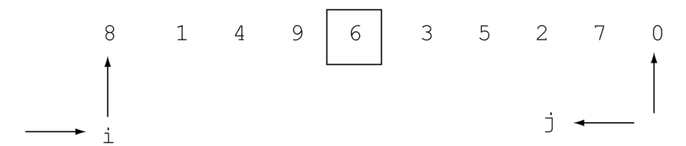
Ejemplo 6.3

Se ordena una lista de números enteros aplicando el algoritmo quicksort*, como pivote se elige el elemento central de la lista.*

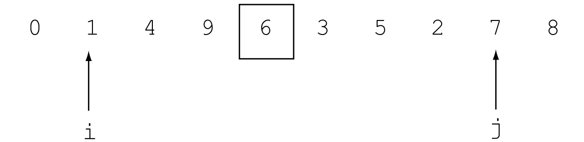
Lista original: `8 1 4 9 6 3 5 2 7 0`
Pivote (elemento central) `6`

La etapa 2 requiere mover todos los elementos menores que el pivote a la parte izquierda del *array* y los elementos mayores a la parte derecha. Para ello, se recorre la lista de izquierda a derecha utilizando un índice `i` que se inicializa a la posición más baja `(inferior)` buscando un elemento mayor al pivote. También se recorre la lista de derecha a izquierda buscando un elemento menor. Para hacer esto se utilizará un índice `j` inicializado a la posición más alta `(superior)`.

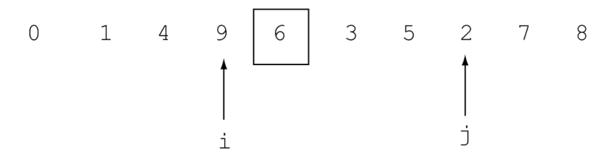
El índice `i` se detiene en el elemento `8` (mayor que el pivote) y el índice `j` se detiene en el elemento `0` (menor que el pivote)



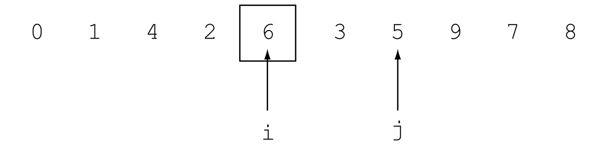
Ahora se intercambian `8` y `0` para que estos dos elementos se sitúen correctamente en cada sublista; y se incrementa el índice `i`, y se decrementa `j` para seguir los intercambios.



A medida que el algoritmo continúa, `i` se detiene en el elemento mayor, `9`, y `j` se detiene en el elemento menor, `2`.



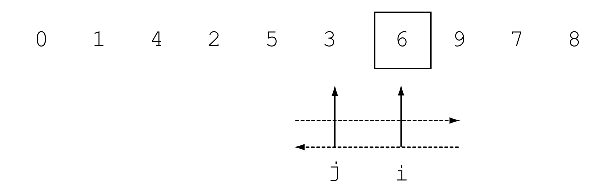
Se intercambian los elementos mientras que `i` y `j` no se crucen. En caso contrario, se detiene este bucle. En el caso anterior, se intercambian `9` y `2`.



Continúa la exploración y ahora el contador `i` se detiene en el elemento `6` (que es el pivote) y el índice `j` se detiene en el elemento menor `5`.

```
0  1  4  2  5  3  [6]  9  7  8
               ↑ ↑
               i j
```

Los índices tienen actualmente los valores `i = 5`, `j = 5`. Continúa la exploración hasta que `i > j`, acaba con `i = 6, j = 5`.



En esta posición, los índices `i` y `j` han cruzado posiciones en el *array*. Se detiene la búsqueda y no se realiza ningún intercambio, ya que el elemento al que accede `j` está ya correctamente situado. Las dos sublistas ya han sido creadas, la lista original se ha dividido en dos particiones:

sublista _ izquierda	pivote	sublista _ derecha
0 1 4 2 5 3	6	9 7 8

El primer problema a resolver en el diseño del algoritmo de *quicksort* es seleccionar el pivote. Aunque su posición, en principio, puede ser cualquiera, una de las decisiones más ponderadas es

aquella que considera el pivote como el elemento central o próximo al central de la lista. La Figura 6.2 muestra las operaciones del algoritmo para ordenar la lista `a[]` de `n` elementos enteros.

Los pasos que sigue el algoritmo *quicksort* son:

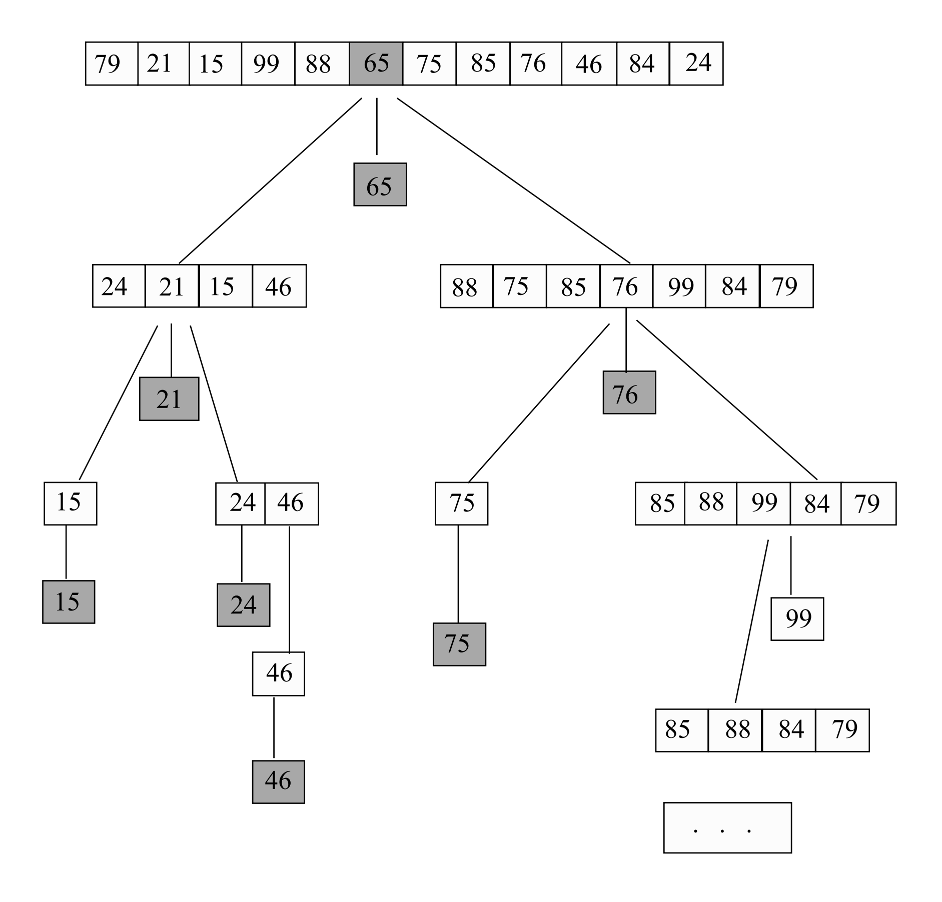
Seleccionar el elemento central de `a[]` como pivote.

Dividir los elementos restantes en particiones *izquierda* y *derecha*, de modo que ningún elemento de la izquierda tenga una clave (valor) mayor que el pivote y que ningún elemento a la derecha tenga una clave menor que la del pivote.

Ordenar la partición izquierda utilizando *quicksort* recursivamente.

Ordenar la partición derecha utilizando *quicksort* recursivamente.

La solución es partición *izquierda* seguida por el pivote y, a continuación, partición *derecha.*



Izquierda: `24, 21, 15, 46`
Pivote: `65`
Derecha: `88, 75, 85, 76, 99, 84, 84, 79`

Figura 6.2 Ordenación rápida eligiendo como pivote el elemento central

6.7.2. Codificación del algoritmo *Quicksort*

La implementación del algoritmo se realiza de manera recursiva; el método `quicksort()` con el argumento *array* `a[]` es, simplemente, la interfaz del algoritmo, su única misión es llamar al método privado del mismo nombre (sobrecargado) `quicksort()` con los argumentos *array* `a[]` y los índices que la delimitan `0` y `a.length-1` (índice inferior y superior).

```
public static void quicksort(double a[])
{
   quicksort(a, 0, a.length-1);
}
```

Y la codificación del método recursivo:

```
private static void quicksort(double a[], int primero, int ultimo)
{

   int i, j, central;
   double pivote;
   central = (primero + ultimo)/2;
   pivote = a[central];
   i = primero;
   j = ultimo;
   do {
      while (a[i] < pivote) i++;
      while (a[j] > pivote) j--;
      if (i <= j)
      {
         intercambiar(a, i, j);
         i++;
         j--;
      }
   }while (i <= j);
   if (primero < j)
      quicksort(a, primero, j); // mismo proceso con sublista izqda

   if (i < ultimo)
      quicksort(a, i, ultimo); // mismo proceso con sublista drcha
}
```

6.7.3. Análisis del algoritmo *Quicksort*

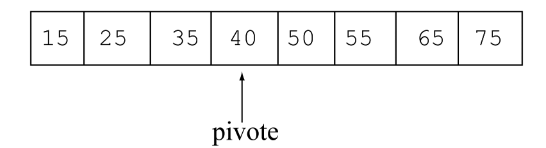
El análisis general de la eficiencia de *quicksort* es difícil. La mejor forma de ilustrar y calcular la complejidad del algoritmo es considerar el número de comparaciones realizadas teniendo en cuenta circunstancias ideales. Supongamos que *n* (número de elementos de la lista) es una potencia de 2, $n = 2^k$ $(k = log_2 n)$. Además, supongamos que el pivote es el elemento central de cada lista, de modo que *quicksort* divide la sublista en dos sublistas aproximadamente iguales.

En la primera exploración o recorrido hay $n-1$ comparaciones. El resultado de la etapa crea dos sublistas aproximadamente de tamaño $n/2$. En la siguiente fase, el proceso de cada sublista requiere de aproximadamente $n/2$ comparaciones. Las comparaciones totales de esta fase son $2*(n/2) = n$. la siguiente fase procesa cuatro sublistas que requieren un total de $4*(n/4)$ comparaciones, etc.

Finalmente, el proceso de división termina después de k pasadas cuando la sublista resultante tenga tamaño 1. El número total de comparaciones es aproximadamente:

$$n + 2*(n/2) + 4*(n/4) + ... + n*(n/n) = n + n + ... + n$$
$$= n * k = n * log_2 n$$

Para una lista normal, la complejidad de *quicksort* es $0(n\ log\ \ n)$. El caso ideal que se ha examinado se realiza realmente cuando la lista (el *array*) está ordenado en orden ascendente. En este caso, el pivote es precisamente el centro de cada sublista.



Si el *array* está en orden ascendente, el primer recorrido encuentra el pivote en el centro de la lista e intercambia cada elemento en las sublistas inferiores y superiores. La lista resultante está casi ordenada, y el algoritmo tiene la complejidad $0(n\ log\ \ n)$.

El escenario del peor caso de *quicksort* ocurre cuando el pivote cae en una sublista de un elemento y deja el resto de los elementos en la segunda sublista. Esto sucede cuando el pivote es siempre el elemento más pequeño de su sublista. En el recorrido inicial, hay n comparaciones y la sublista grande contiene n-1 elementos. En el siguiente recorrido, la sublista mayor requiere n-1 comparaciones y produce una sublista de n-2 elementos, etc. El número total de comparaciones es:

$$n + n\text{-}1 + n\text{-}2 + ... + 2 = (n\text{-}1)*(n+2)/2$$

La complejidad es $0(n^2)$. En general, el algoritmo de ordenación tiene como complejidad media $0(n\ log\ n)$, siendo posiblemente el algoritmo más rápido. La Tabla 6.1 muestra las complejidades de los algoritmos empleados en los métodos explicados en el libro.

Tabla 6.1 Comparación de la complejidad de los métodos de ordenación

Método	**Complejidad**
Burbuja	n^2
Inserción	n^2
Selección	n^2
Montículo	$n\ log\ n$
Fusión	$n\ log\ n$
Shell	$n^{3/2}$
Quicksort	$n\ log\ n$

En conclusión, se suele recomendar que, para listas pequeñas, los métodos más eficientes son: inserción y selección, y para listas grandes, *quicksort*. El algoritmo de Shell suele variar mucho su eficiencia en función de la variación del número de elementos por lo que es más difícil que

en los otros métodos proporcionar un consejo eficiente. Los métodos de fusión y por montículo suelen ser muy eficientes para listas muy grandes.

El método de ordenación por montículos, también llamado *HeapSort,* se desarrolla más adelante, en el Capítulo 11 al ser una aplicación de la estructura de datos montículo.

6.8. ORDENACIÓN DE OBJETOS

Los diversos algoritmos que se han estudiado en los apartados anteriores siempre han ordenado *arrays* de un tipo de dato simple: `int`, `double`, `...` La clase `Vector` de Java está diseñada para almacenar objetos de cualquier tipo. En el Ejemplo 6.4 se almacenan objetos de clases diferentes.

Ejemplo 6.4

Crear un objeto `Vector` *con capacidad para 10 elementos. A continuación, asignar objetos de la clase* `Racional`.

La clase `Vector` tiene un constructor con un argumento de tipo entero para inicializar el objeto `Vector` a la capacidad transmitida en el argumento.

```
Vector v = new Vector(10);
```

La clase `Racional` se caracteriza por dos atributos de tipo entero, *numerador* y *denominador,* además de los métodos que describen el comportamiento de los números racionales: suma...

```
class Racional
{
   private int numerador, denominador;
   public Racional() {numerado = 0; denominador = 1; }
   public Racional(int n, int d)throws Exception
   {
      super();
      numerado = n;
      if (d == 0)
      throw new Exception(" Denominador == 0");
      denominador = d;
   }
   ...
}
```

El siguiente fragmento crea objetos `Racional` y se almacenan en el vector:

```
for (int i = 0; i < 10; i++)
   v.addElement(new Racional(5*i%7, 3*i+1));
```

Los elementos de un `Vector` (consultar apartado 3.6) son objetos de cualquier tipo `(Object)`, exactamente referencias a objetos. Ordenar un vector implica, posiblemente, cambiar el orden que ocupan los objetos, según el criterio de clasificación de éstos. Este criterio debe permitir comparar dos objetos y determinar si un objeto es *mayor,* es *menor* o *igual* que otro.

Un `Vector` ordenado, `w`, posee las mismas propiedades de un *array* de tipo simple ordenado: si `i,j` son dos enteros cualesquiera en el rango `o .. w.size()-1`, siendo `i < j`, entonces `w.elementAt(i)` *es menor o igual que* `w.elementAt(j)`.

¿Cómo comparar objetos? ¿Qué criterio seguir para determinar que el objeto `p1` es menor que el objeto `p2` ? Una alternativa consiste en declarar un `interface` con los métodos `menorQue(),menorIgualQue()` …, y que las clases de los objetos que se ordenan implementen el `interface`.

```
interface Comparador
{
   boolean igualQue(Object op2);
   boolean menorQue(Object op2);
   boolean menorIgualQue(Object op2);
   boolean mayorQue(Object op2);
   boolean mayorIgualQue(Object op2);
}
```

Es responsabilidad de la clase que implementa `Comparador` definir el criterio que aplica para *menor* o *mayor*. Por ejemplo, para objetos de la clase `Racional`, la comparación se realiza en función del valor decimal que resulta de dividir numerador por denominador.

A continuación se declara la clase `Racional` con los métodos de la interfaz; los métodos que implementa el *TAD* `Racional`: suma, multiplicación... , puede incorporarlos el lector y se basan en las operaciones matemáticas del mismo nombre:

```
class Racional implements Comparador
{
   private int numerador, denominador;
   public boolean igualQue(Object op2)
   {
      Racional n2 = (Racional) op2;
      return  ((double)numerador / (double)denominador) ==
              ((double)n2.numerador / (double)n2.denominador );
   }
   public boolean menorQue(Object op2)
   {
      Racional n2 = (Racional) op2;
      return ((double)numerador / (double)denominador) <
              ((double)n2.numerador / (double)n2.denominador );
   }
   public boolean menorIgualQue(Object op2)
   {
      Racional n2 = (Racional) op2;
      return ((double)numerador / (double)denominador) <=
              ((double)n2.numerador / (double)n2.denominador );
   }
   public boolean mayorQue(Object op2)
   {
      Racional n2 = (Racional) op2;
      return  ((double)numerador / (double)denominador) >
                 ((double)n2.numerador / (double)n2.denominador );
   }
   public boolean mayorIgualQue(Object op2)
   {
      Racional n2 = (Racional) op2;
```

```
      return ((double)numerador / (double)denominador) >=
               (double)n2.numerador / (double)n2.denominador );
   }
}
```

Ordenación

Una vez establecidas las condiciones para realizar la ordenación, falta por elegir alguno de los algoritmos de ordenación. Debido a su sencillez, se emplea el *algoritmo de la burbuja*. En el caso de un vector con *n* elementos, la ordenación por burbuja requiere hasta *n-1* pasadas. Por cada pasada se comparan los elementos adyacentes y se intercambian sus valores (referencias a objetos) cuando el primer elemento es mayor que el segundo elemento. Al final de cada pasada, el elemento mayor ha *burbujeado* hasta la cima del vector. Las etapas del algoritmo son :

- En la pasada `1` se comparan los elementos adyacentes.
  `(v[0],v[1]),(v[1],v[2]),(v[2],v[3]),...(v[n-2],v[n-1])`
  Se realizan *n-1* comparaciones, por cada pareja `(v[i],v[i+1])` se intercambian si `v[i+1]` *`es menor que v[i]`*.
  Al final de la pasada, el elemento mayor del vector está situado en `v[n-1]`.
- En la pasada `2` se realizan las mismas comparaciones e intercambios, terminando con el elemento de segundo mayor valor en `v[n-2]`.
- El proceso termina con la pasada *n-1,* en la que el elemento más pequeño se almacena en `v[0]`.

El proceso de ordenación puede terminar en la pasada *n-1*, o bien antes, si en un una pasada no se produce intercambio alguno entre elementos del *vector* es porque ya está ordenado, entonces no son necesarias más pasadas. A continuación se codifica el método de ordenación de un vector cuyos elementos son objetos de la clase `Racional`.

```
public static void ordVector (Vector v)
{
   boolean interruptor = true;
   int pasada, j;
   int n = v.size();
   // bucle externo controla la cantidad de pasadas
   for (pasada = 0; pasada < n-1 && interruptor; pasada++)
   {
      interruptor = false;
      for (j = 0; j < n-pasada-1; j++)
      {
         Racional r;
         r = (Racional)v.elementAt(j);
         if (r.mayorQue(v.elementAt(j+1)))
         {
            // elementos desordenados, se intercambian
            interruptor = true;
            intercambiar(v, j, j+1);
         }
      }
   }
}
```

El intercambio de dos elementos del vector es:

```
private static void intercambiar (Vector v, int i, int j)
{
   Object aux = v.elementAt(i);
   v.setElementAt(v.elementAt(j), i);
   v.setElementAt(aux, j);
}
```

6.9. BÚSQUEDA EN LISTAS: BÚSQUEDA SECUENCIAL Y BINARIA

Con mucha frecuencia, los programadores trabajan con grandes cantidades de datos almacenados en *arrays* y registros, por lo que será necesario determinar si un *array* contiene un valor que coincida con un cierto *valor clave*. El proceso de encontrar un elemento específico de un *array* se denomina *búsqueda*. En esta sección se examinarán dos técnicas de búsqueda: *lineal* o *secuencial,* la técnica más sencilla, y *binaria* o *dicotómica,* la más eficiente.

6.9.1. Búsqueda secuencial

La búsqueda secuencial busca un elemento de una lista utilizando un valor destino llamado ***clave.*** En una búsqueda secuencial (a veces llamada ***búsqueda lineal***), los elementos de una lista o vector se exploran (se examinan) en secuencia, uno después de otro. La búsqueda secuencial es necesaria, por ejemplo, si se desea encontrar a la persona cuyo número de teléfono es 958-220000 en un directorio o listado telefónico de su ciudad. Los directorios de teléfonos están organizados alfabéticamente por el nombre del abonado en lugar de por números de teléfono, de modo que deben explorarse todos los números, uno después de otro, esperando encontrar el número 958-220000.

El algoritmo de búsqueda secuencial compara cada elemento del *array* con la *clave* de búsqueda. Dado que el *array* no está en un orden prefijado, es probable que el elemento a buscar pueda ser el primer elemento, el último elemento o cualquier otro. De promedio, al menos, el programa tendrá que comparar la clave de búsqueda con la mitad de los elementos del *array*. El método de búsqueda lineal funcionará bien con *arrays* pequeños o no ordenados.

6.9.2. Búsqueda binaria

La búsqueda secuencial se aplica a cualquier lista. Si la lista está ordenada, la *búsqueda binaria* proporciona una técnica de búsqueda mejorada. Una búsqueda binaria típica es la búsqueda de una palabra en un diccionario. Dada la palabra, se abre el libro cerca del principio, del centro o del final dependiendo de la primera letra de la palabra que busca. Se puede tener suerte y acertar con la página correcta pero, normalmente, no será así y el lector se mueve a la página anterior o posterior del libro. Por ejemplo, si la palabra comienza con `"J"` y se está en la `"L"` se mueve uno hacia atrás. El proceso continúa hasta que se encuentra la página buscada o hasta que se descubre que la palabra no está en la lista.

Una idea similar se aplica en la búsqueda en una lista ordenada. Se sitúa la lectura en el centro de la lista y se comprueba si nuestra clave coincide con el valor del elemento central. Si no se encuentra el valor de la clave, se sitúa uno en la mitad inferior o superior del elemento central de

la lista. En general, si los datos de la lista están ordenados, se puede utilizar esa información para acortar el tiempo de búsqueda.

Ejemplo 6.5

Se desea averiguar si el elemento 225 se encuentra en el conjunto de datos siguiente:

```
a[0]   a[1]   a[2]   a[3]   a[4]   a[5]   a[6]   a[7]
  13     44     75    100    120    275    325    510
```

El punto central de la lista es el elemento `a[3] (100)`. El valor que se busca es `225` que es mayor que `100`; por consiguiente, la búsqueda continúa en la mitad superior del conjunto de datos de la lista, es decir, en la sublista.

```
a[4]   a[5]   a[6]   a[7]
 120   275    325    510
```

Ahora el elemento en la mitad de esta sublista `a[5] (275)`. El valor buscado, `225`, es menor que `275` y, por consiguiente, la búsqueda continúa en la mitad inferior del conjunto de datos de la lista actual, es decir, en la sublista de un único elemento:

```
a[4]
 120
```

El elemento central de esta sublista es el propio elemento `a[4] (120)`. Al ser `225` mayor que `120` la búsqueda debe continuar en una sublista vacía. Se concluye indicando que no se ha encontrado la clave en la lista.

6.9.3. Algoritmo y codificación de la búsqueda binaria

Suponiendo que la lista esté almacenada como un *array,* donde los índices de la lista son `bajo = 0` y `alto = n-1` donde *n* es el número de elementos del *array,* los pasos a seguir serían:

1. Calcular el índice del punto central del *array:*

   `central = (bajo + alto)/2` (división entera)

2. Comparar el valor de este elemento central con la clave:
   - Si `a[central] < clave`, la nueva sublista de búsqueda tiene por valores extremos de su rango `bajo = central+1 .. alto`.
   - Si `clave < a[central]`, la nueva sublista de búsqueda tiene por valores extremos de su rango `bajo .. central-1`.

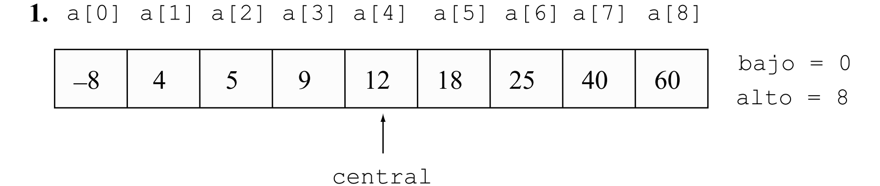
```
          clave                               clave
   [______________________]          [______________________]
bajo          central-1 = alto    bajo = central + 1    alto
```

El algoritmo termina o bien porque se ha encontrado la clave o porque el valor de `bajo` excede a `alto` y el algoritmo devuelve el indicador de fallo de `-1` (búsqueda no encontrada).

Ejemplo 6.6

Sea el array de enteros `A (-8, 4, 5, 9, 12, 18, 25, 40, 60)`*, buscar la clave 40.*

1. `a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8]`



$$\text{central} = \frac{bajo + alto}{2} = \frac{0+8}{2} = 4$$

`clave (40) > a[4] (12)`

2. Buscar en sublista derecha.

18	25	40	60

```
bajo = 5
alto = 8
```

$$\text{central} = \frac{bajo + alto}{2} = \frac{5+8}{2} = 6 \quad \text{(división entera)}$$

`clave (40) > a[6] (25)`

3. Buscar en sublista derecha.

40	60

```
bajo = 7
alto = 8
```

$$\text{central} = \frac{bajo + alto}{2} = \frac{7+8}{2} = 7$$

`clave (40) = a[7] (40)` *búsqueda con éxito*

El algoritmo ha requerido `(3)` comparaciones frente a 8 comparaciones `(n-1, 9-1 = 8)` que se hubieran realizado con la búsqueda secuencial.

Codificación

El método `busquedaBin()` implementa el algoritmo de búsqueda binaria del dato `clave` en un *array* de `length` elementos; el método devuelve la posición que ocupa el elemento, o bien `-1` si no es encontrado.

```
public int busquedaBin(int a[],int clave)
{
```

```
    int central, bajo, alto;
    int valorCentral;
    bajo = 0;
    alto = a.length - 1;
    while (bajo <= alto)
    {
       central = (bajo + alto)/2; // índice de elemento central
       valorCentral = a[central]; // valor del índice central
       if (clave == valorCentral)
          return central; // encontrado, devuelve posición
       else if (clave < valorCentral)
          alto = central -1; // ir a sublista inferior
       else
          bajo = central + 1; // ir a sublista superior
    }
    return -1; //elemento no encontrado
 }
```

6.9.4. Análisis de los algoritmos de búsqueda

Al igual que sucede con las operaciones de ordenación, cuando se realizan operaciones de búsqueda es preciso considerar la eficiencia (complejidad) de los algoritmos empleados en la búsqueda. El grado de eficiencia en una búsqueda será vital cuando se trata de localizar una información en una lista o tabla en memoria, o bien en un archivo de datos.

Complejidad de la búsqueda secuencial

La complejidad de la búsqueda secuencial diferencia entre el comportamiento en el peor y mejor caso. El mejor caso se encuentra cuando aparece una coincidencia en el primer elemento de la lista, por lo que el tiempo de ejecución es $O(1)$. El peor caso se produce cuando el elemento no está en la lista o se encuentra al final de ella. Esto requiere buscar en todos los n términos, lo que implica una complejidad de $O(n)$.

El caso medio requiere un poco de razonamiento probabilista. Para el supuesto de una lista aleatoria es probable que ocurra una coincidencia en cualquier posición. Después de la ejecución de un número grande de búsquedas, la posición media para una coincidencia es el elemento central $n/2$. El elemento central se obtiene después de $n/2$ comparaciones, que definen el coste esperado de la búsqueda. Por esta razón, se dice que la prestación media de la búsqueda secuencial es $O(n)$.

Análisis de la búsqueda binaria

El mejor caso se presenta cuando una coincidencia se encuentra en el punto central de la lista. En este caso, la complejidad es $O(1)$, dado que sólo se realiza una prueba de comparación de igualdad. La complejidad del peor caso es $O(\log_2 n)$, que se produce cuando el elemento no está en la lista o el elemento se encuentra en la última comparación. Se puede deducir intuitivamente esta complejidad. El peor caso se produce cuando se debe continuar la búsqueda y llegar a una sublista de longitud 1. Cada iteración que falla debe continuar disminuyendo la longitud de la sublista por un factor de 2. El tamaño de las sublistas es:

$n \quad n/2 \quad n/4 \quad n/8 \quad \ldots \quad 1$

La división de sublistas requiere *m* iteraciones, y en cada iteración el tamaño de la sublista se reduce a la mitad. La sucesión de tamaños de las sublistas hasta una sublista de longitud 1 será:

$n \quad n/2 \quad n/2^2 \quad n/2^3 \quad n/2^4 \; .... \quad n/2^m$

siendo $n/2^m = 1$. Tomando logaritmos en base 2 a la expresión anterior tendríamos:

$n = 2^m$
$m = log_2\, n$

Por esa razón, la complejidad del peor caso es $O(log_2 n)$. Cada iteración requiere una operación de comparación:

$Total\ comparaciones \approx 1 + log_2 n$

Comparación de la búsqueda binaria y secuencial

La comparación en tiempo entre los algoritmos de búsqueda secuencial y binaria se va haciendo espectacular a medida que crece el tamaño de la lista de elementos. Hay que tener presente que, en el caso de la búsqueda secuencial, en el peor de los casos coincidirá el número de elementos examinados con el número de elementos de la lista, tal como representa su complejidad *O(n)*.

Sin embargo, en el caso de la búsqueda binaria, hay que tener presente, por ejemplo, que $2^{10} = 1.024$, lo cual implica el examen de 11 posibles elementos; si se aumenta el número de elementos de una lista a 2.048 y teniendo presente que $2^{11} = 2.048$ implicará que el número máximo de elementos examinados en la búsqueda binaria es 12. Si se sigue este planteamiento, se puede encontrar el número *m* más pequeño para una lista de 1.000.000, tal que:

$2^n \geq 1.000.000$

Es decir, $2^{19} = 524.288$, $2^{20} = 1.048.576$ y, por tanto, el número de elementos examinados (en el peor de los casos) es 21.

Tabla 6.2 Comparación de las búsquedas binaria y secuencial

Números de elementos examinados		
Tamaño de la lista	**Búsqueda binaria**	**Búsqueda secuencial**
1	1	1
10	4	10
1.000	11	1.000
5.000	14	5.000
100.000	18	100.000
1.000.000	21	1.000.000

La Tabla 6.2 muestra la comparación de los métodos de búsqueda secuencial y búsqueda binaria. En la misma tabla se puede apreciar una comparación del número de elementos que se deben examinar utilizando búsqueda secuencial y binaria. Esta tabla muestra la eficiencia de la búsqueda binaria comparada con la búsqueda secuencial, cuyos resultados de tiempo vienen

dados por las funciones de complejidad $O(log_2n)$ y $O(n)$ de las búsquedas binaria y secuencial respectivamente.

A tener en cuenta

La búsqueda secuencial se aplica para localizar una clave en un vector no ordenado. Para aplicar el algoritmo de búsqueda binaria, la lista o vector donde se busca debe de estar ordenado. La complejidad de la búsqueda binaria es logarítmica, `O(Log n)`, más eficiente que la búsqueda secuencial que tiene complejidad lineal, `O(n)`.

RESUMEN

- Una de las aplicaciones más frecuentes, en programación es la ordenación.
- Existen dos técnicas de ordenación fundamentales en gestión de datos: *ordenación de listas* y *ordenación de archivos.*
- Los datos se pueden ordenar en orden ascendente o en orden descendente.
- Cada recorrido de los datos durante el proceso de ordenación se conoce como *pasada* o *iteración.*
- Los algoritmos de ordenación básicos son:
  - Selección.
  - Inserción.
  - Burbuja.
- Los algoritmos de ordenación más avanzados son:
  - *Shell.*
  - *Heapsort* (por montículos).
  - *Mergesort.*
  - *Radixsort.*
  - *Binsort.*
  - *Quicksort.*
- La eficiencia de los algoritmos de burbuja, inserción y selección es $0(n^2)$.
- La eficiencia de los algoritmos heapsort, radixsort, mergesort y quicksort es $0(n \, Log \, n)$.
- La búsqueda es el proceso de encontrar la posición de un elemento destino dentro de una lista.
- Existen dos métodos básicos de búsqueda en arrays: **búsqueda secuencial** y **binaria.**
- La **búsqueda secuencial** se utiliza normalmente cuando el array no está ordenado. Comienza en el principio del array y busca hasta que se encuentra el dato buscado y se llega al final de la lista.
- Si un array está ordenado, se puede utilizar un algoritmo más eficiente denominado **búsqueda binaria.**
- La eficiencia de una búsqueda secuencial es $0(n)$.
- La eficiencia de una búsqueda binaria es $0(log \, n)$.

EJERCICIOS

6.1. ¿Cuál es la diferencia entre ordenación por intercambio y ordenación por el método de la burbuja?

6.2. Se desea eliminar todos los números duplicados de una lista o vector (array). Por ejemplo, si el array toma los valores

```
4    7    11    4    9    5    11    7    3    5
```

ha de cambiarse a

```
4    7    11    9    5    3
```

Escribir un método que elimine los elementos duplicados de un array.

6.3. Escribir un método que elimine los elementos duplicados de un vector ordenado. ¿Cuál es la eficiencia del método? Compare la eficiencia con la que tiene el método del Ejercicio 6.2.

6.4. Un vector contiene los elementos mostrados a continuación. Los primeros dos elementos se han ordenado utilizando un algoritmo de inserción. ¿Cuál será el valor de los elementos del vector después de tres pasadas más del algoritmo?

3	13	8	25	45	23	98	58

6.5. Dada la siguiente lista

47	3	21	32	56	92

Después de dos pasadas de un algoritmo de ordenación, el array se ha quedado dispuesto así.

3	21	47	32	56	92

¿Qué algoritmo de ordenación se está utilizando (selección, burbuja o inserción)? Justifique la respuesta.

6.6. Un array contiene los elementos indicados más abajo. Utilizando el algoritmo de ordenación Shell, encuentre las pasadas y los intercambios que se realizan para su ordenación.

8	43	17	6	40	16	18	97	11	7

6.7. Partiendo del mismo array que en el Ejercicio 6.6, encuentre las particiones e intercambios que realiza el algoritmo de ordenación *Quicksort* para su ordenación.

6.8. Un array de registros se quiere ordenar según el campo clave *fecha de nacimiento*. Dicho campo consta de tres subcampos: día, mes y año, de 2, 2 y 4 dígitos respectivamente. Adaptar el método de ordenación *Radixsort* a esta ordenación.

6.9. Un array contiene los elementos indicados más abajo. Utilizando el algoritmo de búsqueda binaria, trazar las etapas necesarias para encontrar el número 88.

8	13	17	26	44	56	88	97

Hacer la misma búsqueda pero para el número 20.

6.10. Escribir la función de ordenación correspondiente al método *Radixsort* para poner en orden alfabético una lista de *n* nombres.

6.11. Escribir una función de búsqueda binaria aplicado a un array ordenado descendentemente.

6.12. Supongamos que se tiene una secuencia de *n* números que deben ser clasificados:

1. Utilizando el método de Shell, ¿cuántas comparaciones y cuántos intercambios se requieren para clasificar la secuencia si
   - ya está clasificado?
   - está en orden inverso?
2. Repetir el paso 1 para el método de *Quicksort.*

PROBLEMAS

6.1. Un método de ordenación muy simple, pero no muy eficiente, de elementos x_1, x_2, x_3,...x_n en orden ascendente es el siguiente:

Paso 1: Localizar el elemento más pequeño de la lista x_1 a x_n; intercambiarlo con x_1.

Paso 2: Localizar el elemento más pequeño de la lista x_2 a x_n, intercambiarlo con x_2.

Paso 3: Localizar el elemento más pequeño de la lista x_3 a x_n, intercambiarlo con x_n.

En el último paso, los dos últimos elementos se comparan e intercambian, si es necesario, y la ordenación se termina. Escribir un programa para ordenar una lista de elementos, siguiendo este método.

6.2. Dado un vector *x* de *n* elementos reales, donde *n* es impar, diseñar un método que calcule y devuelva la mediana de ese vector. La mediana es el valor tal que la mitad de los números son mayores que él y la otra mitad son menores. Escribir un programa de prueba.

6.3. Se trata de resolver el siguiente problema escolar: dadas las notas de los alumnos de un colegio en el primer curso de bachillerato en las diferentes asignaturas (5, por comodidad), se trata de calcular la media de cada alumno, la media de cada asignatura, la media total de la clase y ordenar los alumnos por orden decreciente de notas medias individuales.

Nota: utilizar como algoritmo de ordenación el método Shell.

6.4. Escribir un programa de consulta de teléfonos. Leer un conjunto de datos de 1.000 nombres y números de teléfono de un archivo que contenga los números en orden aleatorio. Las consultas han de poder realizarse por nombre y por número de teléfono.

6.5. Realizar un programa que compare el tiempo de cálculo de las búsquedas secuencial y binaria. Una lista A se rellena con 2.000 enteros aleatorios en el rango 0 ... 1999 y a continuación se ordena. Una segunda lista B se rellena con 500 enteros aleatorios en el mismo rango. Los elementos de B se utilizan como claves de los algoritmos de búsqueda.

6.6. Se dispone de dos vectores, *Maestro* y *Esclavo,* del mismo tipo y número de elementos. Se deben imprimir en dos columnas adyacentes. Se ordena el vector *Maestro*, pero siempre que un elemento de *Maestro* se mueva, el elemento correspondiente de *Esclavo* debe moverse también; es decir, cualquier cosa que se haga a *Maestro*[*i*] debe hacerse a *Esclavo*[*i*]. Después de realizar la ordenación, se imprimen de nuevo los vectores. Escribir un programa que realice esta tarea.

Nota: utilizar como algoritmo de ordenación el método *Quicksort.*

6.7. Cada línea de un archivo de datos contiene información sobre una compañía de informática. La línea contiene el nombre del empleado, las ventas efectuadas por el mismo y el número de años de antigüedad del empleado en la compañía. Escribir un programa que lea la información del archivo de datos y, a continuación, se visualiza. La información debe ser ordenada por ventas de mayor a menor y visualizada de nuevo.

6.8. Se desea realizar un programa que realice las siguientes tareas:

a) Generar, aleatoriamente, una lista de 999 de números reales en el rango de 0 a 2000.
b) Ordenar en modo creciente por el método de la burbuja.
c) Ordenar en modo creciente por el método Shell.
d) Ordenar en modo creciente por el método *Radixsort.*
e) Buscar si existe el número x (leído del teclado) en la lista. La búsqueda debe ser binaria.

Ampliar el programa anterior de modo que pueda obtener y visualizar en el programa principal los siguientes tiempos:

t1. Tiempo empleado en ordenar la lista por cada uno de los métodos.
t2. Tiempo que se emplearía en *ordenar* la lista ya ordenada.
t3. Tiempo empleado en ordenar la lista ordenada en orden inverso.

6.9. Construir un método que permita ordenar por fechas y de mayor a menor un vector de *n* elementos que contiene datos de contratos (n <= 50). Cada elemento del vector debe ser un objeto con los campos día, mes, año y número de contrato. Pueden existir diversos contratos con la misma fecha, pero no números de contrato repetidos.

Nota. El método a utilizar para ordenar será el de *Radixsort.*

6.10. Escribir un programa que genere un vector de 10.000 números aleatorios del 1 al 500. Realizar la ordenación del vector por dos métodos:

- *Binsort.*
- *Radixsort.*

Escriba el tiempo empleado en la ordenación de cada método.

6.11. Se leen dos listas de números enteros, A y B, de 100 y 60 elementos, respectivamente. Se desea resolver las siguientes tareas:

a) Ordenar aplicando el método de *Quicksort* cada una de las listas A y B.

b) Crear una lista C por intercalación o mezcla de las listas A y B.

c) Visualizar la lista C ordenada.



CAPITULO 7

Algoritmos de ordenación de archivos

Objetivos

Con el estudio de este capítulo, usted podrá:

- Manejar archivos como objetos de una clase.
- Conocer la jerarquía de clases definida en el entorno de Java para el manejo de archivos.
- Distinguir entre ordenación en memoria y ordenación externa.
- Conocer los algoritmos de ordenación de archivos basados en la mezcla.
- Realizar la ordenación de archivos secuenciales con mezcla múltiple.
- Conocer la base de la ordenación polifásica.

Contenido

Conceptos clave

- Acceso secuencial.
- Archivos de texto.
- Flujos.
- Memoria externa.
- Memoria interna.
- Mezcla.
- Ordenación.
- Organización de un archivo.
- Persistencia de objetos.
- Secuencia de Fibonacci
- Secuencia ordenada.

.

INTRODUCCIÓN

Los algoritmos de ordenación de *arrays* o vectores están limitados a una secuencia de datos relativamente pequeña, ya que los datos se guardan en memoria interna. No se pueden aplicar si la cantidad de datos a ordenar no cabe en la memoria principal de la computadora y como consecuencia, están almacenados en un dispositivo de memoria externa, tal como un disquete o un disco óptico. Los archivos tienen como finalidad guardar datos de forma permanente, una vez que acaba la aplicación los datos siguen disponibles para que otra aplicación pueda recuperarlos para su consulta o modificación. Es necesario aplicar nuevas técnicas de ordenación que se complementen con las ya estudiadas. Entre las más importantes destaca la *mezcla*. Mezclar, significa combinar dos (o más) secuencias en una sola secuencia ordenada por medio de una selección repetida entre los componentes accesibles en ese momento.

El proceso de archivos en Java se hace mediante el concepto de **flujo** *(streams)* o canal, también denominado secuencia. Los flujos pueden estar abiertos o cerrados, conducen los datos entre el programa y los dispositivos externos. Con las clases y sus métodos proporcionadas por el *paquete* de clases `java.io` se pueden tratar archivos secuenciales, de acceso directo, archivos indexados...

En este capítulo se revisa el tratamiento de archivos en Java y se analizan los métodos de ordenación de archivos, ordenación externa, más populares y eficaces.

7.1. FLUJOS Y ARCHIVOS

Un ***fichero*** *(archivo)* de datos —o, simplemente, un **archivo**— es una colección de registros relacionados entre sí, con aspectos en común y organizados para un propósito específico. Por ejemplo, un fichero de una clase escolar contiene un conjunto de registros de los estudiantes de esa clase. Otros ejemplos, el fichero de nóminas de una empresa, el fichero de inventario, stocks, etc.

Un archivo en una computadora es una estructura diseñada para contener datos. Los datos están organizados de tal modo que puedan ser recuperados fácilmente, actualizados o borrados y almacenados de nuevo en el archivo con todos los cambios realizados.

Según las características del soporte empleado y el modo en que se han organizado los registros, se consideran dos tipos de acceso a los registros de un archivo:

- *Acceso secuencial.*
- *Acceso directo.*

El *acceso secuencial* implica el acceso a un archivo según el orden de almacenamiento de sus registros, uno tras otro.

El *acceso directo* implica el acceso a un registro determinado, sin que ello implique la consulta de los registros precedentes. Este tipo de acceso sólo es posible con soportes direccionales.

En Java, un archivo es, sencillamente, una secuencia de bytes, que son la representación de los datos almacenados. Java dispone de clases para trabajar las secuencias de bytes como datos de tipos básicos `(int, double, String...)`; incluso, para escribir o leer del archivo objetos. El diseño del archivo es el que establece la forma de manejar las secuencias de bytes, con una organización secuencial, o bien de acceso directo.

Un **flujo** *(stream)* es una abstracción que se refiere a un *flujo* o *corriente* de datos que fluyen entre un origen o fuente *(productor)* y un destino o sumidero *(consumidor)*. Entre el origen y el destino debe existir una conexión o canal *(pipe)* por la que circulen los datos. La apertura de un archivo supone establecer la conexión del programa con el dispositivo que contiene al archivo, por el canal que comunica el archivo con el programa van a fluir las secuencias de datos. Abrir

un archivo supone crear un objeto que queda asociado a un flujo. Al comenzar la ejecución de un programa, en Java se crean automáticamente tres objetos flujo, son tres canales por los que pueden *fluir datos,* de entrada o de salida. Estos son objetos definidos en la clase `System`:

System.in; Objeto entrada estándar; permite la entrada al programa de flujos de bytes desde el teclado.

System.out; Objeto salida estándar; permite al programa la salida de datos por pantalla.

System.err; Objeto para salida estándar de errores; permite al programa la salida de errores por pantalla.

En Java, un archivo es simplemente un flujo externo, una secuencia de bytes almacenados en un dispositivo externo (normalmente en disco). Si el archivo se abre para salida, es un flujo de archivo de salida. Si el archivo se abre para entrada, es un flujo de archivo de entrada. Los programas leen o escriben en el flujo, que puede estar conectado a un dispositivo o a otro. El flujo es, por tanto, una abstracción, de tal forma que las operaciones que realizan los programas son sobre el flujo independientemente del dispositivo al que esté asociado.

A tener en cuenta

El paquete `java.io` agrupa el conjunto de clases e interfaces necesarias para procesar archivos. Es necesario utilizar clases de este paquete, por lo que se debe incorporar al programa con la sentencia `import java.io.*`.

7.2. CLASE `File`

Un archivo consta de un nombre, además de la ruta que indica donde está ubicado, por ejemplo, `"C:\pasaje.dat"`. Este identificador del archivo (cadena de caracteres) se transmite al constructor del flujo de entrada o de salida que procesa al fichero:

```
FileOutputStream f = new  FileOutputStream("C:\pasaje.dat");
```

Los constructores de flujos que esperan un archivo están sobrecargados para, además de recibir el archivo como cadena, recibir un objeto de la clase `File`. Este tipo de objeto contiene el nombre del archivo, la ruta y más propiedades relativas al archivo.

La clase `File` define métodos para conocer propiedades del archivo (*última modificación, permisos de acceso, tamaño* ...); también, métodos para modificar alguna característica del archivo.

Los constructores de `File` permiten inicializar el objeto con el nombre de un archivo y la ruta donde se encuentra. También permiten inicializar el objeto con otro objeto `File` como ruta y el nombre del archivo.

```
public File(String nombreCompleto)
```
Crea un objeto File con el nombre y ruta del archivo pasado como argumento.

```
public File(String ruta, String nombre)
```
Crea un objeto File con la ruta y el nombre del archivo pasado como argumentos.

```
public File(File ruta, String nombre)
```
Crea un objeto File con un primer argumento que a su vez es un objeto File con la ruta y el nombre del archivo como segundo argumento.

Por ejemplo:

```
File miFichero = new File("C:\LIBRO\Almacen.dat");
```

crea un objeto `File` con el archivo `Almacen.dat` que está en la ruta `C:\LIBRO`.

```
File otro = new File("COCINA", "Enseres.dat");
```

> **Nota**
>
> Es una buena práctica crear objetos `File` con el archivo que se va a procesar, para pasar el objeto al constructor del flujo en vez de pasar directamente el nombre del archivo. De esta forma se pueden hacer controles previos sobre el archivo.

7.3. FLUJOS Y JERARQUÍA DE CLASES

Todo el proceso de entrada y salida en Java se hace a través de flujos *(stream).* En los programas hay que crear objetos *stream* y, en muchas ocasiones, hacer uso de los objetos `in`, `out` de la clase `System`. Los flujos de datos, de caracteres de bytes se pueden clasificar en flujos de entrada y en flujos de salida. En consonancia, Java declara dos clases (derivan directamente de la clase `Object`): `InputStream` y `OutputStream`. Ambas son clases abstractas que declaran métodos que deben de redefinirse en sus clases derivadas. `InputStream` es la clase base de todas las clases definidas para flujos de entrada, y `OutputStream` es la clase base de todas las clases definidas para flujos de salida. La Tabla 7.1 muestra las clase derivadas mas importantes de éstas.

Tabla 7.1 Primer nivel de la jerarquía de clases de Entrada/Salida

InputStream	OutputStream
`FileInputStream`	`FileOutputStream`
`PipedInputStream`	`PipedOutputStream`
`ObjectInputStream`	`ObjectOutputStream`
`StringBufferInputStream`	`FilterOutputStream`
`FilterInputStream`	

7.3.1. Archivos de bajo nivel: `FileInputStream` y `FileOutputStream`

Todo archivo, tanto para entrada como salida, se puede considerar como una secuencia de bytes. A partir de estas secuencias de bytes, *flujos de bajo nivel*, se construyen flujos de más alto nivel para proceso de datos complejos, desde tipos básicos hasta objetos. Las clases `FileInputStream` y `FileOutputStream` se utilizan para leer o escribir bytes en un archivo; objetos de estas dos clases son los flujos de entrada y salida, respectivamente, a nivel de bytes. Los constructores de ambas clases permiten crear flujos (objetos) asociados a un archivo que se encuentra en cualquier

dispositivo, de modo que el archivo queda abierto. Por ejemplo, el flujo `mf` se asocia al archivo `Temperatura.dat` del directorio por defecto:

```
FileOutputStream mf = new FileOutputStream("Temperatura.dat");
```

Las operaciones que a continuación se realicen con `mf` escriben secuencias de bytes en el archivo `Temperatura.dat`.

La clase `FileInputStream` dispone de métodos para leer un byte o una secuencia de bytes. A continuación se escriben los métodos mas importantes de esta clase, todos con visibilidad `public`; es importante tener en cuenta la excepción que pueden *lanzar* para que cuando se invoquen se haga un tratamiento de la excepción.

```
FileInputStream(String nombre) throws FileNotFoundException;
```
Crea un objeto inicializado con el nombre de archivo que se pasa como argumento.

```
FileInputStream(File nombre) throws FileNotFoundException;
```
Crea un objeto inicializado con el objeto archivo pasado como argumento.

```
int read() throws IOException;
```
Lee un byte del flujo asociado. Devuelve –1 si alcanza el fin del fichero.

```
int read(byte[] s) throws IOException;
```
Lee una secuencia de bytes del flujo y se almacena en el array s. Devuelve –1 si alcanza el fin del fichero, o bien el número de bytes leídos.

```
int read(byte[] s, int org, int len) throws IOException;
```
Lee una secuencia de bytes del flujo y se almacena en array s desde la posición org y un máximo de len bytes. Devuelve –1 si alcanza el fin del fichero, o bien el númerode bytes leídos.

```
void close()throws IOException;
```
Cierra el flujo,el archivo queda disponible para posterior uso.

La clase `FileOutputStream` dispone de métodos para escribir bytes en el flujo de salida asociado a un archivo. Los constructores inicializan objetos con el nombre del archivo, o bien con el archivo como un objeto `File`, el archivo queda abierto. A continuación se escriben los constructores y métodos mas importantes, todos con visibilidad `public`.

```
FileOutputStream(String nombre) throws IOException;
```
Crea un objeto inicializado con el nombre de archivo que se pasa como argumento.

```
FileOutputStream(String nombre, boolean sw) throws IOException;
```
Crea un objeto inicializado con el nombre de archivo que se pasa como argumento. En el caso de que sw = true los bytes escritos se añaden al final.

```
FileOutputStream(File nombre) throws IOException;
```
Crea un objeto inicializado con el objeto archivo pasado como argumento.

```
void write(byte a) throws IOException;
```
Escribe el byte a *en el flujo asociado.*

```
void write(byte[] s) throws IOException;
```
Escribe el array de bytes en el flujo.

```
void write(byte[] s, int org, int len) throws IOException;
```
Escribe array s desde la posición org y un máximo de len bytes en el flujo.

```
void close()throws IOException;
```
Cierra el flujo, el archivo queda disponible para posterior uso.

Nota de programación

Una vez creado un flujo, pueden realizarse operaciones típicas de archivos: *leer* (flujos de entrada), *escribir* (flujos de salida). Es el constructor el encargado de abrir el flujo, en definitiva, de abrir el archivo. Si el constructor no puede crear el flujo (archivo de lectura no existe…), lanza la excepción `FileNotFoundException`.

Siempre que finaliza la ejecución de un programa en Java se cierran los flujos abiertos. Sin embargo, se aconseja ejecutar el método `close()` cuando deje de utilizarse un flujo, de esa manera se liberan recursos asignados y el archivo queda disponible.

Ejemplo 7.1

Dado el archivo `jardines.txt`, *se desea escribir toda su información en el archivo* `jardinOld.txt`.

El primero archivo se asocia con un flujo de entrada, el segundo fichero con un flujo de salida. Entonces, se instancia un objeto *flujo de entrada* y otro de salida del tipo `FileInputStream` y `FileOutputStream` respectivamente. La lectura se realiza byte a byte con el método `read()`; cada byte se escribe en el flujo de salida invocando al método `write()`. El proceso termina cuando `read()` devuelve `-1`, señal de haber alcanzado el fin del archivo.

```
import java.io.*;

public class CopiaArchivo
{
   public static void main(String [] a)
   {

      FileInputStream origen = null;
      FileOutputStream destino = null;
      File f1 = new File("jardines.txt");
      File f2 = new File("jardinOld.txt");
      try
      {
         origen = new FileInputStream(f1);
         destino = new FileOutputStream(f2);
         int c;
         while ((c = origen.read()) != -1)
            destino.write((byte)c);
      }
      catch (IOException er)
      {
         System.out.println("Excepción en los flujos "
                           + er.getMessage());
      }
      finally {
         try
         {
            origen.close();
            destino.close();
         }
         catch (IOException er)
         {
            er.printStackTrace();
         }
      }
   }
}
```

7.4. ORDENACIÓN DE UN ARCHIVO. MÉTODOS DE ORDENACIÓN EXTERNA

Los algoritmos de ordenación interna, en memoria principal, utilizan *arrays* para guardar los elementos a ordenar; es necesario que la memoria interna tenga capacidad suficiente. Para ordenar secuencias grandes de elementos, que posiblemente no pueden almacenarse en memoria, se aplican los *algoritmos de ordenación externa*. La ordenación externa está ligada con los archivos y los dispositivos en que se encuentra, al leer el archivo para realizar la ordenación el tiempo de lectura de los registros es notablemente mayor que el tiempo que se tarda en realizar las operaciones de ordenación.

Un archivo está formado por una secuencia de `n` elementos; cada elemento es un objeto (registro). Los objetos, *R(i),* pueden ser comparables si disponen de una clave, *K(i),* mediante la cual se pueden hacer comparaciones. El archivo está ordenado respecto a la clave si:

$$\forall\ i < j \Rightarrow K(i) < K(j)$$

La ordenación de los objetos (registros) de una archivo mediante archivos auxiliares se denomina **ordenación externa.** Los distintos algoritmos de ordenación externa utilizan el esquema general de separación en tramos y fusión o mezcla. Por separación se entiende la distribución de secuencias de registros ordenados en varios archivos; por fusión, la mezcla de dos o más secuencias ordenadas en una única secuencia ordenada. Variaciones de este esquema general dan lugar a diferentes algoritmos de ordenación externa.

A tener en cuenta

El tiempo de un algoritmo de ordenación de registros de un archivo, *ordenación externa,* depende notablemente del dispositivo de almacenamiento. Los algoritmos repiten consecutivamente una fase de *separación* en tramos y otra de *mezcla* que da lugar a tramos ordenados cada vez mas largos; se considera únicamente el acceso secuencial a los registros (objetos).

7.5. MEZCLA DIRECTA

Es el método mas simple de ordenación externa, utiliza el esquema iterativo de separar secuencias de registros y su mezcla. Se opera con el archivo original y dos archivos auxiliares. El proceso consiste en:

1. Separar los registros individuales del archivo original *O* en dos archivos, *F1* y *F2*.
2. Mezclar los archivos *F1* y *F2* combinando registros aislados (según sus claves) y formando pares ordenados que son escritos en el archivo *O*.
3. Separar pares de registros del archivo original O en los dos archivos auxiliares *F1* y *F2*.
4. Mezclar *F1* y *F2* combinando pares de registros y formando cuádruplos ordenados que son escritos en el archivo *O*.
5. Se repiten los pasos de separación y mezcla, combinando cuádruplos para formar óctuplos ordenados. En cada paso de separación y mezcla se duplica el tamaño de las subsecuencias mezcladas, así hasta que la longitud de la subsecuencia sea la que tiene el archivo, y en ese momento el archivo original *O* está ordenado.

A recordar

El método de ordenación externa *mezcla directa* en el paso *i* obtiene secuencias ordenadas de longitud 2^i. Termina cuando la longitud de la secuencia es igual al número de registros del archivo.

Ejemplo 7.2

En este ejemplo se hace un seguimiento del algoritmo de ordenación externa mezcla directa. Se supone un archivo formado por registros que tienen un campo clave de tipo entero; las claves son las siguientes:

```
34  23  12  59  73  44  8  19  28  51
```

Se realizan los pasos del algoritmo de mezcla directa para ordenar la secuencia. Se considera el archivo O como el original y F1 y F2 como archivos auxiliares.

Pasada 1

Separación:

```
F1: 34  12  73  8  28
F2: 23  59  44  19  51
```

Mezcla formando duplos ordenados:

```
O: 23  34  12  59  44  73  8  19  28  51
```

Pasada 2

Separación de duplos:

```
F1: 23  34  44  73  28  51
F2: 12  59  8  19
```

Mezcla formando cuádruplos ordenados:

```
O: 12  23  34  59  8  19  44  73  28  51
```

Pasada 3

Separación de cuádruplos:

```
F1: 12  23  34  59  28  51
F2:  8  19  44  73
```

Mezcla formando óctuplos ordenados:

```
O: 8  12  19  23  34  44  59  73  28  51
```

Pasada 4

Separación de octuplos:

```
F1: 8  12  19  23  34  44  59  73
F2: 28  51
```

Mezcla con la que ya se obtiene el archivo ordenado:

```
O: 8  12  19  23  28  34  44  51  59  73
```

En el ejemplo han sido necesarias cuatro pasadas, cada una constituye una fase de separación y otra de mezcla.

Después de `i` pasadas se tiene el archivo `O` con subsecuencias ordenadas de longitud 2^i, si $2^i \geq n$, siendo `n` el número de registros, el archivo estará ordenado. El número de pasadas que realiza el algoritmo se obtiene tomando logaritmos, $i \geq Log_2 n$, $\lceil Log\ n \rceil$ pasadas serán suficientes. Cada pasada escribe el total `n` de registros, por lo que el número total de movimientos es `O(n Log n)`.

El tiempo de las comparaciones realizadas en la fase de fusión es insignificante respecto a las operaciones de movimiento de registros en los archivos externos, por esa razón no resulta interesante analizar el número de comparaciones.

7.5.1. Codificación del algoritmo de mezcla directa

Los métodos `distribuir()` y `mezclar()`implementan las dos partes fundamentales del algoritmo. El primero *separa* secuencias de registros del archivo original en los dos archivos auxiliares, el segundo *mezcla* secuencias de los dos archivos auxiliares, y la secuencia resultante se escribe en el archivo original En cada pasada se realiza una llamada a los métodos `distribuir()` y `mezclar()`, obteniendo una secuencia del doble de longitud de registros ordenados. El algoritmo termina cuando la longitud de la secuencia iguala al número de registros.

A continuación se escribe un programa en el que se codifican estos dos métodos. El método `distribuir()` utiliza un método auxiliar, `subsecuencia()`, para escribir una secuencia de un número especificado de registros en un archivo auxiliar. Este método se llama alternativamente con los archivos `F1` y `F2`, de esa forma se consigue distribuir el archivo en secuencias de longitud fija. En la codificación del método `mezclar()` es necesario controlar la lectura del registro (dato de tipo entero) para detectar la marca de *fin de fichero*.

Los flujos que se utilizan son de tipo `DataInputStream` y `DataOutputStream` para poder leer o escribir directamente datos de tipo entero. En primer lugar y con el fin de probar el algoritmo, se genera un secuencia de valores enteros aleatoriamente.

```
import java.io.*;

class OrdenExtMzclaDirecta
{
   static final int N = 716;
   static final int TOPE = 999;

   public static void main(String []a)
   {
      File f = new File("ArchivoOrigen");
      DataOutputStream flujo = null;
      try {
            // se genera un archivo secuencialmente de claves enteras
         flujo = new DataOutputStream(
                  new BufferedOutputStream(new FileOutputStream(f)));
         for (int j = 1; j <= N; j++)
            flujo.writeInt((int)(1+TOPE*Math.random()));
         flujo.close();
         mezclaDirecta(f);
      }
      catch (IOException e)
```

```
    {
       System.out.println("Error entrada/salida durante proceso" +
                                              " de ordenación ");
       e.printStackTrace();
    }
    escribir(f);
 }

 static void mezclaDirecta(File f) throws IOException
 {
    int longSec;
    int numReg;
    File f1 = new File("ArchivoAux1");
    File f2 = new File("ArchivoAux2");
       /* número de registros se obtiene dividiendo el tamaño
          del archivo por el tamaño del registro: 4.
       */
    numReg = (int)f.length()/4;
    longSec = 1;
    while (longSec < numReg)
    {
       distribuir(f, f1, f2, longSec, numReg);
       mezclar(f1, f2, f, longSec, numReg);
       longSec *= 2;
    }
 }

 static
 void distribuir(File f, File f1, File f2,
                 int longSec, int numReg) throws IOException
 {
    int numSec, resto, i;
    DataInputStream flujo = new DataInputStream(
                new BufferedInputStream(new FileInputStream(f)));
    DataOutputStream flujo1 = new DataOutputStream(
             new BufferedOutputStream(new FileOutputStream(f1)));
    DataOutputStream flujo2 = new DataOutputStream(
             new BufferedOutputStream(new FileOutputStream(f2)));
    numSec = numReg /(2*longSec);
    resto =  numReg %(2*longSec);
    //distribuye secuencias de longitud longSec
    for (i = 1; i <= numSec; i++)
    {
       subSecuencia(flujo, flujo1, longSec);
       subSecuencia(flujo, flujo2, longSec);
    }
    /*
       Se procesa el resto de registros del archivo
    */
    if (resto > longSec)
       resto -= longSec;
    else
    {
       longSec = resto;
       resto = 0;
    }
    subSecuencia(flujo, flujo1, longSec);
```

```
      subSecuencia(flujo, flujo2, resto);
      flujo.close();
      flujo1.close();
      flujo2.close();
   }

   private static
   void subSecuencia(DataInput f, DataOutput t,
                              int longSec) throws IOException
   {
      int clave;
      //escribe en el flujo t el dato entero leído de f
      for (int j = 1; j <= longSec; j++)
      {
         clave = f.readInt();
         t.writeInt(clave);
      }
   }

   static void
   mezclar(File f1, File f2, File f,
                int lonSec, int numReg) throws IOException
   {
      int numSec, resto, i, j, k;
      int clave1 = 0, clave2 = 0;

      numSec = numReg /(2*lonSec); // número de subsecuencias
      resto = numReg %(2*lonSec);
      DataOutputStream flujo = new DataOutputStream(
                new BufferedOutputStream(new FileOutputStream(f)));
      DataInputStream flujo1 = new DataInputStream(
             new BufferedInputStream(new FileInputStream(f1)));
      DataInputStream flujo2 = new DataInputStream(
             new BufferedInputStream(new FileInputStream(f2)));
      //claves iniciales
      clave1 = flujo1.readInt();
      clave2 = flujo2.readInt();
      //bucle para controlar todo el proceso de mezcla
      for (int s = 1; s <= numSec+1; s++)
      {
         int n1, n2;
         n1 = n2 = lonSec;
         if (s == numSec+1)
         {  // proceso de subsecuencia incompleta
            if (resto > lonSec)
               n2 = resto - lonSec;
            else
            {
               n1 = resto;
               n2 = 0;
            }
         }
         i = j = 1;
         while (i <= n1 && j <= n2)
         {
            int clave;
            if (clave1 < clave2)
```

```
         {
            clave = clave1;
            try {
               clave1 = flujo1.readInt();}
            catch(EOFException e){;}
            i++;
         }
         else
         {
            clave = clave2;
            try {
               clave2 = flujo2.readInt();}
            catch(EOFException e){;}
            j++;
         }
         flujo.writeInt(clave);
      }
      /*
         Los registros no procesados se escriben directamente
      */
      for (k = i; k <= n1; k++)
      {
         flujo.writeInt(clave1);
         try {
            clave1 = flujo1.readInt();}
         catch(EOFException e){;}
      }
      for (k = j; k <= n2; k++)
      {
         flujo.writeInt(clave2);
         try {
            clave2 = flujo2.readInt();}
         catch(EOFException e){;}
      }
   }
   flujo.close();
   flujo1.close();
   flujo2.close();
}

static  void escribir(File f)
{
   int clave, k;
   boolean mas = true;
   DataInputStream flujo = null;
   try {
      flujo = new DataInputStream(
            new BufferedInputStream(new FileInputStream(f)));
      k = 0;
      System.out.println("ARCHIVO DE CLAVES TIPO INT");
      while (mas)
      {
         k++;
         System.out.print(flujo.readInt() + " ");
         if (k % 11 == 0) System.out.println();
      }
   }
```

```
        catch (EOFException eof)
        {
           System.out.println("\n *** Fin del archivo ***\n");
           try
           {
              flujo.close();
           }
           catch (IOException er)
           {
              er.printStackTrace();
           }

        }
        catch (IOException e)
        {
           e.printStackTrace();
        }
     }
  }
```

7.6. FUSIÓN NATURAL

El método de fusión natural mejora el tiempo de ejecución de la mezcla directa al introducir una pequeña variación respecto a la longitud de las secuencias de registros. En el método de mezcla directa, las secuencias de registros tienen longitudes que son múltiplos de dos: `1, 2, 4, 8, 16...` de tal forma que el número de *pasadas* a realizar es fijo, dependiente del número de registros. La mezcla directa no tiene en cuenta la circunstancia de que pueda haber, *de manera natural,* secuencias más largas ya ordenadas que también puedan mezclarse y dar lugar a otra secuencia ordenada.

A recordar

El método de ordenación externa *fusión natural,* distribuye en todo momento secuencias ordenadas (tramos) lo más largas posibles y mezcla secuencias ordenadas lo más largas posibles.

7.6.1. Algoritmo de la fusión natural

La característica fundamental de este método es la mezcla de secuencias ordenadas máximas, o simplemente tramo máximo. Una secuencia ordenada a_i ... a_j es tal que:

$a_k \leq a_{k+1}$ para $k = i \ldots j-1$

$a_{i-1} > a_i$

$a_j > a_{j+1}$

Por ejemplo, en esta lista de claves enteras: `4 9 11 5 8 12 9 17 18 21 26 18` los tramos máximo que se encuentran: `4 9 11;` **`5`** `8 12;` **`9`** `17 18 21 26;` **`18`**. La ruptura de un tramo ocurre cuando la clave actual es menor que la clave anterior.

El método de fusión natural funde tramos máximos en lugar de secuencias de tamaño fijo y predeterminado. Esto hace que se optimice el número de pasadas a realizar, disminuyendo en el método de fusión natural. Los tramos tienen la propiedad de si se tienen dos listas de `n` tramos cada una y se mezclan, producir una lista de `n` tramos exactamente; como consecuencia, el número total de tramos disminuye, al menos se divide por dos, en cada pasada del algoritmo de ordenación fusión natural.

Ejemplo 7.3

Un archivo está formado por registros que tienen un campo clave de tipo entero, suponiendo que éstas son las siguientes:

```
17 31 5 59 33 41 43 67 11 23 29 47 3 7 71 2 19 57 37 61
```

En el ejemplo se van a realizar los pasos que sigue el algoritmo de fusión natural para ordenar la secuencia. Se considera el archivo O como el original, F1 y F2 como archivos auxiliares.

Los tramos máximos de la lista de claves se separan por el carácter ´:

```
17 31´ 5 59´ 33 41 43 67´ 11 23 29 47´ 3 7 71´ 2 19 57´ 37 61
```

Pasada 1

Separación:

```
F1: 17 31  33 41 43 67´ 3  7 71´ 37 61
F2:  5 59´ 11 23 29 47´ 2 19 57
```

Se puede observar que, de manera natural, al distribuirse por tramos, la secuencia `17 31` se ha expandido junto a `33 41 43 67` y han formado una única secuencia ordenada.

Mezcla o fusión de tramos:

```
O: 5 17 31 33 41 43 59 67´ 3 7 11 23 29 47 71´ 2 19 37 57 61
```

Pasada 2

Separación:

```
F1: 5 17 31 33 41 43 59 67´ 2 19 37 57 61
F2: 3  7 11 23 29 47 71
```

Mezcla o fusión de tramos:

```
O: 3 5 7 11 17 23 29 31 33 41 43 47 59 67 71´ 2 19 37 57 61
```

Pasada 3

Separación:

```
F1: 3 5 7 11 17 23 29 31 33 41 43 47 59 67 71
F2: 2 19 37 57 61
```

Mezcla o fusión de tramos máximos:

```
O: 2 3 5 7 11 17 19 23 29 31 33 37 41 43 47 57 59 61 67 71
```

La lista ya está ordenada, la longitud del tramo máximo es igual al número de registros del archivo. En el ejemplo han sido necesarias tres pasadas, cada una constituye una fase de separación y otra de mezcla o fusión.

El algoritmo, al igual que en el método mezcla directa, se puede escribir descomponiendo las acciones que realiza en dos rutinas: `separarNatural()` y `mezclaNatural()`. La primera *separa* tramos máximos del archivo original en los dos archivos auxiliares. La segunda *mezcla* tramos máximos de los dos archivos auxiliares y escribe el resultado en el archivo original. El algoritmo termina cuando hay un único tramo; entonces el archivo está ordenado.

```
Ordenación MezclaNatural
inicio
   repetir
      separarNatural(f, f1, f2)
      mezclaNatural(f, f1, f2, numeroTramos)
   hasta numerosTramos = 1
fin
```

7.7. MEZCLA EQUILIBRADA MÚLTIPLE

La eficiencia de los métodos de ordenación externa es directamente proporcional al número de pasadas. Para aumentar la eficiencia hay que reducir el número de pasadas, de esa forma se reduce el número de operaciones de entrada/salida en dispositivos externos. Se observa que el método de *fusión natural* reduce el número de pasadas respecto a la mezcla directa; ambos métodos tienen en común que se utilizan dos archivos auxiliares, además de las dos fases: *separación* y *mezcla*.

Otra forma de reducir el número de *pasadas* es incrementando el número de archivos auxiliares. Supóngase que se tienen `w` tramos distribuidos equitativamente en `m` archivos. La primera *pasada* mezcla los `w` tramos y da lugar a `w/m` tramos. En la siguiente *pasada*, la mezcla de los `w/m` tramos da lugar a w/m^2 tramos; en la siguiente, se reducen a w/m^3; después de `i` *pasadas,* quedarán w/m^i tramos.

Para determinar la complejidad del algoritmo mediante mezcla de *m-uples* tramos, se supone que, *en el peor de los casos,* un archivo de `n` registros tiene `n` tramos iniciales; entonces, el número de pasadas necesarias para la ordenación completa es $[Log_m n]$, como cada pasada realiza `n` operaciones de entrada/salida con los registros, la eficiencia es $O(nLog_m n)$. La mejora obtenida en cuanto a la disminución de las pasadas necesarias para la ordenación hace que los movimientos o transferencias de cada registro sea $Log_2 m$ veces menor.

La sucesión del número de tramos, suponiendo tanto tramos iniciales como registros, sería:

$$n,\ n/m,\ n/m^2,\ n/m^3 \ldots n/m^t = 1$$

tomando logaritmos en base `m`, se calcula el número de tramos `t`:

$$n = m^t;\ Log_m\, n = Log_m\, m^t \Rightarrow t = Log_m\, n$$

7.7.1. Algoritmo de la mezcla equilibrada múltiple

La mezcla equilibrada múltiple utiliza `m` archivos auxiliares, de los que `m/2` son de entrada y `m/2` de salida. Inicialmente, se distribuyen los tramos del archivo de origen en los `m/2` archivos auxiliares. A partir de esta distribución, se repiten los procesos de mezcla reduciendo a la mitad

el número de tramos hasta que queda un único tramo. De esta forma, el proceso de mezcla se realiza en una sola fase en lugar de las dos fases *(separación, fusión)* de los algoritmos mezcla directa y fusión natural. Los pasos que sigue el algoritmo son:

1. Distribuir registros del archivo original por tramos en los `m/2` primeros archivos auxiliares. A continuación, estos se consideran archivos de entrada.
2. Mezclar tramos de los `m/2` archivos de entrada y escribirlos consecutivamente en los `m/2` archivos de salida.
3. Cambiar la finalidad de los archivos, los de entrada pasan a ser de salida y viceversa; repetir a partir del segundo paso hasta que quede un único tramo, entonces la secuencia está ordenada.

7.7.2. Declaración de archivos para la mezcla equilibrada múltiple

La principal variación, en cuanto a las variables que se utilizan, está en que los flujos correspondientes a los archivos auxiliares se agrupan en un *array*. Los registros del archivo a ordenar tienen un campo clave ordinal, respecto al cual se realiza la ordenación. La constante `N` representa el número de archivos auxiliares (flujos); la constante `N2` es el número de archivos de entrada, la mitad de `N`, también es el índice inicial de los flujos de salida (en Java, un *array* se indexa con base cero).

```
static final int N = 6;
static final int N2 = N/2;
File []f = new File[N];
```

La variable `f[]` representa los archivos auxiliares, alternativamente la primera mitad y la segunda irán cambiando su cometido, entrada o salida.

7.7.3. Cambio de finalidad de un archivo: *entrada* ↔ *salida*

La forma de cambiar la finalidad de los archivos (*entrada* ↔ *salida*) se hace mediante una tabla de correspondencia entre índices de archivo, de tal forma que, en lugar de acceder a un archivo por el índice del *array,* se accede por la tabla, la cual cambia alternativamente los índices de los archivos y, de esa forma, pasan, alternativamente, de ser de entrada a ser de salida *(flujos de entrada, flujos de salida).*

`int[] c = new int[N];` *Tabla de índices de archivo.*

Inicialmente, `c[i] = i  ∀i ∈ 0 .. N-1.`

Como consecuencia, los archivos de entrada son:

```
f[c[0]], f[c[1]],…, f[c[N2-1]];
```

y los ficheros de salida son la otra mitad:

```
f[c[N2]], f[c[N2+1]], … f[c[N-1]]
```

Para realizar el cambio de archivo de entrada por salida, se intercambian los valores de las dos mitades de la tabla de correspondencia:

```
c[0]   ↔ c[N2]
c[1]   ↔ c[N2+1]
   . . .
c[N2] ↔ c[N-1]
```

En definitiva, con la tabla `c[]` siempre se accede de igual forma a los archivos, lo que cambia son los índices que contiene `c[]`.

Al mezclar tramos de los archivos de entrada no se alcanza el fin de tramo en todos los archivos al mismo tiempo. Un tramo termina cuando es *fin de archivo* (excepción `EOFException`), o bien cuando la siguiente clave es menor que la actual; en cualquier caso, el archivo que le corresponde ha de quedar inactivo. Para despreocuparse de si el archivo está activo o no, se utiliza otro *array* cuyas posiciones indican si el archivo correspondiente al índice está activo. Como en algún momento del proceso de mezcla no todos los archivos de entrada están activos, en otra tabla de correspondencia `cd[]`, sólo para archivos de entrada, se tienen en todo momento los índices de los archivos de entrada activos (en los que no se ha alcanzado el *fin de fichero*).

> ***Nota de programación***
>
> En Java, se procesan los archivos mediante flujos. Entonces, en la codificación se trabaja con flujos de entrada y flujos de salida asociados a los correpondientes archivos.

7.7.4. Control del número de tramos

El primer paso del algoritmo realiza la distribución de los tramos del archivo original en los archivos de entrada, a la vez determina el número de tramos del archivo. En todo momento es importante conocer el número de tramos, ya que cuando queda sólo uno el archivo está ordenado, y éste será el archivo `f[c[0]]`.

En la ejecución del método de ordenación llega un momento en que el número de tramos a mezclar va a ser menor que el número de archivos de entrada. La variable `k1` contiene el número de archivos de entrada. Cuando el número de tramos, `t`, es mayor o igual que la mitad de los archivos, el valor de `k1` es justamente la mitad; en caso de ser `t` menor que dicha mitad, `k1` será igual a `t` y, por último, cuando `t` es `1`, `k1` también es `1` y el archivo `f[c[0]]` está ya ordenado.

7.7.5. Codificación del algoritmo de mezcla equilibrada múltiple

El programa supone, por simplicidad, que el archivo que se ordena está formado por registros de tipo entero. La clase `MzclaMultiple` declara un atributo `File` con el archivo origen, `fo`, y un *array* de referencias a `File`, `f[]`, con los archivos auxiliares. Las operaciones de entrada/salida se realizan con flujos `DataInputStream` y `DataOutputStream` y sus respectivos métodos `readInt()`, `writeInt()`. El proceso de mezcla de un tramo de cada uno de los `m/2` archivos (flujos) de entrada comienza leyendo de cada uno de los flujos un registro (clave de tipo entero) y guardándolo en un *array* de registros `rs[]`. La mezcla selecciona repetidamente el registro

menor, con una llamada al método `minimo()`; el proceso de selección tiene en cuenta que los registros, que se encuentran en el *array* `rs[]`, se correspondan con archivos activos. Cada vez que es seleccionado un registro, se lee el siguiente registro del mismo flujo de entrada, a no ser que haya acabado el tramo; así hasta que se terminan de mezclar todos los tramos. El método `fintramos()` determina si se ha llegado al final de todos los tramos que están involucrados en la mezcla.

La codificación que se presenta a continuación se realiza con `6` archivos auxiliares. En primer lugar, se crea el archivo original con números enteros generados aleatoriamente con llamadas al método `Math.random()`, a continuación, se llama al método que implementa el algoritmo de ordenación *mezcla equilibrada múltiple*.

```
import java.io.*;

class MzclaMultiple
{
   static final int N = 6;
   static final int N2 = N/2;
   static File f0;
   static File []f = new File[N];
   static final int NumReg = 149;
   static final int TOPE = 999;

   public static void main(String []a)
   {
      String[] nomf = {"ar1", "ar2", "ar3", "ar4", "ar5", "ar6"};
      f0 = new File("ArchivoOrigen");
      for (int i = 0; i < N; i++)
         f[i] = new File(nomf[i]);

      DataOutputStream flujo = null;
      // se genera un archivo secuencialmente de claves enteras
      try {
         flujo = new DataOutputStream(
                 new BufferedOutputStream(new FileOutputStream(f0)));

         for (int i = 1; i <= NumReg; i++)
            flujo.writeInt((int)(1+TOPE*Math.random()));
         flujo.close();
         System.out.print("Archivo original ...  ");
         escribir(f0);
         mezclaEqMple();
      }
      catch (IOException e)
      {
         System.out.println("Error entrada/salida durante proceso" +
                                                 " de ordenación ");
         e.printStackTrace();
      }
   }
   //método de ordenación
   public static void mezclaEqMple()
   {
      int i, j, k, k1, t;
      int anterior;
      int [] c = new int[N];
      int [] cd = new int[N];
      int [] r = new int[N2];
```

```
Object [] flujos = new Object[N];
DataInputStream flujoEntradaActual = null;
DataOutputStream flujoSalidaActual = null;
boolean [] actvs = new boolean[N2];
// distribución inicial de tramos desde archivo origen
try {
   t = distribuir();
   for (i = 0; i < N; i++)
      c[i] = i;
   // bucle hasta número de tramos == 1: archivo ordenado
   do {
      k1 = (t < N2) ? t : N2;
      for (i = 0; i < k1; i++)
      {
         flujos[c[i]] = new DataInputStream(
         new BufferedInputStream(new FileInputStream(f[c[i]])));
         cd[i] = c[i];
      }
      j = N2 ; // índice de archivo de salida
      t = 0;
      for (i = j; i < N; i++)
         flujos[c[i]] = new DataOutputStream(
                              new BufferedOutputStream(
                                    new FileOutputStream(f[c[i]])));
      // entrada de una clave de cada flujo
      for (int n = 0; n < k1; n++)
      {
         flujoEntradaActual = (DataInputStream)flujos[cd[n]];
         r[n] = flujoEntradaActual.readInt();
      }

      while (k1 > 0)
      {
         t++; // mezcla de otro tramo
         for (i = 0; i < k1; i++)
            actvs[i] = true;
         flujoSalidaActual = (DataOutputStream)flujos[c[j]];
         while (!finDeTramos(actvs, k1))
         {
            int n;
            n = minimo(r, actvs, k1);
            flujoEntradaActual = (DataInputStream)flujos[cd[n]];
            flujoSalidaActual.writeInt(r[n]);
            anterior = r[n];
            try {
               r[n] = flujoEntradaActual.readInt();
               if (anterior > r[n]) // fin de tramo
                  actvs[n] = false;
            }
            catch (EOFException eof)
            {
               k1--;
               flujoEntradaActual.close();
               cd[n] = cd[k1];
               r[n] = r[k1];
               actvs[n] = actvs[k1];
               actvs[k1] = false;// no se accede a posición k1
            }
         }
```

```
            j = (j < N-1) ? j+1 : N2; // siguiente flujo de salida
         }

         for (i = N2; i < N; i++)
         {
            flujoSalidaActual = (DataOutputStream)flujos[c[i]];
            flujoSalidaActual.close();
         }
         /*
            Cambio de finalidad de los flujos: entrada<->salida
         */
         for (i = 0; i < N2; i++)
         {
            int a;
            a       = c[i];
            c[i]    = c[i+N2];
            c[i+N2] = a;
         }

      } while (t > 1);
      System.out.print("Archivo ordenado ...  ");
      escribir(f[c[0]]);
   }
   catch (IOException er)
   {
      er.printStackTrace();
   }

}
//distribuye tramos de flujos de entrada en flujos de salida
private static int distribuir() throws IOException
{
   int anterior, j, nt;
   int clave;

   DataInputStream flujo = new DataInputStream(
              new BufferedInputStream(new FileInputStream(f0)));
   DataOutputStream [] flujoSalida = new DataOutputStream[N2];
   for (j = 0; j < N2; j++)
   {
      flujoSalida[j] = new DataOutputStream(
      new BufferedOutputStream(new FileOutputStream(f[j])));
   }

   anterior = -TOPE;
   clave = anterior + 1;
   j = 0;     // indice del flujo de salida
   nt = 0;
   // bucle termina con la excepción fin de fichero
   try {
      while (true)
      {
         clave = flujo.readInt();
         while (anterior <= clave)
         {
            flujoSalida[j].writeInt(clave);
            anterior = clave;
            clave = flujo.readInt();
         }
```

```
            nt++;                       // nuevo tramo
            j = (j < N2-1) ? j+1 : 0;   // siguiente archivo
            flujoSalida[j].writeInt(clave);
            anterior = clave;
         }
      }
      catch (EOFException eof)
      {
         nt++;   // cuenta ultimo tramo
         System.out.println("\n*** Número de tramos: " + nt + " ***");
         flujo.close();
         for (j = 0; j < N2; j++)
            flujoSalida[j].close();
         return nt;
      }
   }
   //devuelve el índice del menor valor del array de claves
   private static int minimo(int [] r, boolean [] activo, int n)
   {
      int i, indice;
      int m;

      i = indice = 0;
      m = TOPE+1;

      for ( ; i < n; i++)
      {
         if (activo[i] && r[i] < m)
         {
            m = r[i];
            indice = i;
         }
      }
      return indice;
   }
   //devuelve true si no hay tramo activo
   private static boolean finDeTramos(boolean [] activo, int n)
   {
      boolean s = true;

      for(int k = 0; k < n; k++)
      {
         if (activo[k])    s = false;
      }
      return s;
   }
   //escribe las claves del archivo
   static  void escribir(File f)
   {
      int clave, k;
      boolean mas = true;
      DataInputStream flujo = null;
      try {
         flujo = new DataInputStream(
                 new BufferedInputStream(new FileInputStream(f)));
         k = 0;
         while (mas)
         {
            k++;
            System.out.print(flujo.readInt() + " ");
```

```
            if (k % 19 == 0) System.out.println();
         }
      }
      catch (IOException eof)
      {
         System.out.println("\n *** Fin del archivo ***\n");
         try
         {
            if (eof instanceof EOFException)
               flujo.close();
         }
         catch (IOException er)
         {
            er.printStackTrace();
         }

      }

   }

}
```

7.8. MÉTODO POLIFÁSICO DE ORDENACIÓN EXTERNA

La estrategia seguida en el método de mezcla equilibrada múltiple emplea `2m` archivos para ordenar `n` registros, de tal forma que si los `n` registros están distribuidos en `m` tramos, en una *pasada* (distribución + mezcla) quedan ordenados. La utilización de `2m` archivos puede hacer que el número de éstos sea demasiado elevado y no todas las aplicaciones puedan soportarlo. La mezcla equilibrada múltiple puede mejorarse consiguiendo ordenar `m` tramos con sólo `m+1` archivos; para ello hay que abandonar la idea rígida de *pasada* en la que la finalidad de los archivos de entrada no cambia hasta que se leen todos.

El *método polifásico* utiliza `m` archivos auxiliares para ordenar `n` registros de un archivo. La característica que marca la diferencia de este método respecto a los otros es que continuamente se consideran `m-1` archivos de entrada, desde los que se mezclan registros, y un archivo de salida. En el momento en que uno de los archivos de entrada alcanza su final hay un cambio de cometido, pasa a ser considerado como archivo de salida, y el archivo que en ese momento era de salida pasa a ser de entrada y la mezcla de tramos continúa. La sucesión de pasadas continúa hasta alcanzar el archivo ordenado.

Cabe recordar la propiedad base de todos los métodos de mezcla: *la mezcla de k tramos de los archivos de entrada se transforma en k tramos en el archivo de salida*.

A tener en cuenta

La mezcla polifásica se caracteriza por realizar una mezcla continua de tramos, de tal forma que si se utilizan `m` archivos auxiliares, en un momento dado uno de ellos es archivo de salida y los otros `m-1` archivos son de entrada. Durante el proceso, cuando se alcanza el *registro de fin de archivo* en un archivo de entrada, este pasa a ser de salida, el anterior archivo de salida pasa a ser de entrada y la mezcla continua. La dificultad del método es que el número de tramos iniciales debe pertenecer a una sucesión de números dependiente de `m`.

7.8.1. Mezcla polifásica con `m = 3` archivos

A continuación se muestra un ejemplo en el que se supone un archivo original de `55` tramos. El número de archivos auxiliares es `m = 3`; entonces, en todo momento `2` archivos son de entrada y un tercero de salida.

Inicialmente, el método distribuye los tramos, de manera no uniforme, en los archivos `F1`, `F2`. Suponiendo que se dispone de `55` tramos, se sitúan `34` en `F1` y `21` en `F2`; el archivo `F3` es, inicialmente, de salida. Empieza la mezcla, y `21` tramos de `F1` se fusionan con los `21` tramos de `F2` dando lugar a `21` tramos en `F3`. La situación en este momento es que en `F1` quedan `13` tramos, en `F2` se ha alcanzado el *fin de fichero* y en `F3` hay `21` tramos; `F2` pasa a ser archivo de salida y la mezcla continua entre `F1` y `F3`. Ahora se mezclan `13` tramos de `F1` con `13` tramos de `F3`, dando lugar a `13` tramos que se van escribiendo en `F2`, se ha alcanzado *el fin de fichero* de `F1`. En el archivo `F2` hay `13` tramos y en el `F3` quedan `8` tramos, y continúa la mezcla con `F1` como archivo de salida. En la nueva pasada se mezclan `8` tramos que se escriben en `F1`, quedan `5` tramos en `F2` y ninguno en `F3` ya que se ha alcanzado *el fin de fichero*. El proceso sigue hasta que queda un único tramo y el fichero ha quedado ordenado. La Tabla 7.2 muestra las sucesivas pasadas y los tramos de cada archivo hasta que termina la ordenación

Tabla 7.2 Tramos en cada archivo después de cada pasada en la mezcla polifásica con 3 archivos

		Después de cada pasada							
	Tramos iniciales	F1+F2	F1+F3	F2+F3	F1+F2	F1+F3	F2+F3	F1+F2	F1+F3
F1	34	13	0	8	3	0	2	1	0
F2	21	0	13	5	0	3	1	0	1
F3	0	21	8	0	5	2	0	1	0

Es evidente que se ha partido de una distribución óptima. Además, si se escribe la sucesión de tramos mezclados a partir de la distribución inicial, `21, 13, 8, 5, 3, 2, 1, 1`, es justamente la sucesión de los números de Fibonacci:

$$f_{i+1} = f_i + f_{i-1} \quad \forall i \geq 1 \text{ tal que } f_1 = 1,\ f_0 = 0$$

Entonces, si el número de tramos iniciales f_k es tal que es un número de Fibonacci, la mejor forma de hacer la distribución inicial es según la sucesión de Fibonacci, f_{k-1} y f_{k-2}. Sin embargo, el archivo de origen no siempre dispone de un número de tramos perteneciente a la sucesión de Fibonacci, y en esos casos se recurre a escribir tramos ficticios para conseguir un número de la secuencia de Fibonacci.

Distribución para m = 4 archivos

Se puede extender el proceso de mezcla polifásica a un número mayor de archivos. Por ejemplo, si se parte de un archivo origen con `31` tramos y se utilizan `m = 4` archivos para la mezcla polifásica, la distribución inicial y los tramos de cada archivo se muestran en la Tabla 7.3.

Tabla 7.3 Tramos en cada archivo en la mezcla polifásica con 4 archivos

		Después de cada pasada				
	Tramos iniciales	F1+F2+F3	F1+F2+F4	F1+F3+F4	F2+F3+F4	F1+F2+F3
F1	13	6	2	0	1	0
F2	11	4	0	2	1	0
F3	7	0	4	2	1	0
F4	0	7	3	1	0	1

La Tabla 7.4 muestra en forma tabular la distribución perfecta de los tramos en los archivos para cada pasada pero numeradas en orden inverso.

Tabla 7.4 Tramos que intervienen en cada pasada para 3 tramos

L	$t_1^{L)}$	$t_2^{L)}$	$t_3^{L)}$
5	13	11	7
4	7	6	4
3	4	3	2
2	2	2	1
1	1	1	1
0	1	0	0

Observando los datos de la Tabla 7.4 se deducen ciertas relaciones entre t_1, t_2, t_3 en un nivel:

$$t_3^{L+1)} = t_1^{L)}$$
$$t_2^{L+1)} = t_1^{L)} + t_3^{L)}$$
$$t_1^{L+1)} = t_1^{L)} + t_2^{L)}$$
$$\forall\ L > 0 \quad y\ t_1^{0)} = 1\ ,\ t_2^{0)} = 0,\ t_3^{0)} = 0$$

Estas relaciones van a ser muy útiles para encontrar la distribución inicial ideal cuando se desee ordenar un archivo de un número arbitrario de tramos. Además, haciendo el cambio de variable de f_i por $t_1^{L)}$ se tiene la sucesión de los números de Fibonacci de orden 2:

$$f_{i+1} = f_i + f_{i-1} + f_{i-2}\ \forall i \geq 2 \text{ tal que } f_2 = 1,\ f_1 = 0,\ f_0 = 0$$

A tener en cuenta

En general, la sucesión de *números de fibonacci de orden k* tiene la expresión:

$$f_{i+1} = f_i + f_{i-1} + \ldots + f_{i-k+1}\ \forall i \geq k-1 \text{ tal que,}$$
$$f_{k-1} = 1,\ f_{k-2} = 0\ \ldots,\ f_0 = 0$$

7.8.2. Distribución inicial de tramos

En los dos ejemplos expuestos se ha aplicado el método polifásico de manera ideal, la distribución inicial de tramos era perfecta en función del número de archivos auxiliares y de la correspondiente sucesión de Fibonacci. Las fórmulas recurrentes que permiten obtener el número de tramos para cada pasada `L`, en el supuesto de utilizar `m` archivos, son las siguientes:

$$t_{m-1}^{L+1)} = t_1^{L)}$$
$$t_{m-2}^{L+1)} = t_1^{L)} + t_{m-1}^{L)}$$
$$\ldots$$
$$t_2^{L+1)} = t_1^{L)} + t_3^{L)}$$
$$t_1^{L+1)} = t_1^{L)} + t_2^{L)}$$
$$\forall\ L > 0 \quad y\ t_1^{0)} = 1\ ,\ t_i^{0)} = 0 \quad \forall i \leq m-1$$

Con estas relaciones puede conocerse de antemano el número de tramos necesarios para aplicar el método polifásico con `m` archivos. Es evidente que no siempre el número de tramos iniciales del archivo a ordenar va a coincidir con ese número ideal, ¿qué hacer? La respuesta es sencilla: se simulan los tramos necesarios para completar la distribución perfecta con tramos vacíos o tramos ficticios. ¿Cómo hay que tratar los tramos ficticios? La selección de un tramo ficticio de un archivo i es, simplemente, ignorar el archivo y, por consiguiente, desecharlo de la mezcla del tramo correspondiente. En el supuesto de que el tramo sea ficticio para los `m-1` archivos de entrada, no habrá que hacer ninguna operación, simplemente considerar un tramo ficticio en el archivo de salida. Los distribución inicial ha de repartir los tramos fícticios lo más uniformemente posible en los `m-1` archivos.

Para tener en cuenta estas consideraciones relativas a los tramos, se utilizan dos *arrays,* `a[]` y `d[]`. El primero, contiene los números de tramos que ha de tener cada archivo de entrada en una pasada dada; el segundo, guarda el número de tramos ficticios que tiene cada archivo.

El proceso se inicia de *abajo a arriba;* por ejemplo, consideremos la Tabla 7.4, se empieza asignando al *array* `a[]` el numero de tramos correspondientes con la última mezcla, siempre `(1,1,... ,1)`. A la vez, al *array* de tramos ficticios `d[]` es también `(1,1,... ,1)`, de tal forma que cada vez que se copie un tramo, del archivo origen, en el archivo `i`, se decrementa `d[i]`, y así con cada uno de los `m-1` archivos.

Si no se ha terminado el archivo original, se determina de nuevo el número de tramos, pero del siguiente nivel, y los tramos que hay que añadir a cada archivo para alcanzar ese segundo nivel según las relaciones recurrentes. Los tramos ficticios de cada archivo, `d[i]`, coincidirán con el número de tramos que se deben añadir, `a[i]`, para que, posteriormente, según se vayan añadiendo tramos al archivo origen, se vaya decrementando `d[i]`.

Ejemplo 7.7

Se desea ordenar un archivo que dispone de 28 tramos, y se van a utilizar m = 4 archivos auxiliares. Encontrar la distribución inicial de tramos en los m-1 archivos.

La distribución se realiza consecutivamente, de nivel a nivel, según la Tabla 7.4. El primer nivel consta de tres tramos, `a(1,1,1)`, se distribuyen 3 tramos; se alcanza un segundo nivel con 5 tramos, `a(2,2,1)`, y como ya se distribuyeron 3 tramos se añaden 2 nuevos tramos, `(2,2,1)-(1,1,1) = (1,1,0)` y el *array* de tramos ficticios `d[]` se inicializa a los mismos valores, `(1,1,0)`. Una vez completados, los tramos para este segundo nivel, `d[]` se queda `a (0,0,0)`, se pasa al

siguiente nivel. Los tramos para alcanzar el tercer nivel son 9, a(4,3,2), entonces se debe añadir (4,3,2)-(2,2,1) = (2,1,1), y se inician los tramos ficticios d(2,1,1); se han distribuido 4 nuevos tramos.

Sucesivamente, se calcula el número de tramos de cada nuevo nivel y los tramos que hay que añadir para obtener esa cantidad; esos tramos serán, inicialmente, los tramos ficticios. A medida que se reparten tramos se decrementa el correspondiente elemento del *array* de tramos ficticios d[]. La Tabla 7.5 muestra los distintos valores que toman a[] y d[] en cada nivel, hasta completar los 28 tramos.

En el supuesto planteado, una vez completado el cuarto nivel se han repartido 17 tramos, y se pasa al siguiente; ahora a = (13,11,7), los tramos incorporados son (13,11,7)-(7,6,4) = = (6,5,3), y los tramos ficticios siempre se inicializan al número de tramos que se añaden, en este nivel d(6,5,3). En el supuesto de 28 tramos, restan únicamente 28-17 = 11 tramos, que se distribuyen uniformemente en los archivos, y al finalizar la distribución el *array* de tramos ficticios queda d = (2,1,0) que se deben tener en cuenta, en los archivos correspondientes, durante el siguiente paso, que es la mezcla.

Cuantos más tramos haya más niveles se alcanzan. En cualquier caso, una vez terminado el proceso de distribución del archivo original, se tiene en a[]el número de tramos para ese nivel, y en d[] el número de tramos ficticios que tiene cada archivo, necesario para el proceso de mezcla.

Tabla 7.5 Distribución inicial con 28 tramos y m = 4 archivos

	Tramos iniciales			**Tramos nivel, añadir**	
		Número de nivel			
		2	3	4	5
a[]	(1,1,1)	(2,2,1)	(4,3,2)	(7,6,4)	**(13,11,7)**
	añadir	(1,0,0)	(2,1,1)	(3,3,2)	(6,5,3)
a[]	(1,1,1)	(2,1,0)	(2,1,1)	(3,3,2)	(6,5,3)
	(0,0,0)	(0,0,0)	(0,0,0)	(0,0,0)	**(2,1,0)**

7.8.3. Algoritmo de la mezcla

Una vez realizada la distribución inicial en los m-1 primeros archivos auxiliares, se repite el proceso de mezcla hasta que queda un único tramo. El número de pasadas es conocido de antemano, ha sido calculado durante la distribución, es el número de niveles alcanzado; también se conoce la distribución inicial de los tramos, que se encuentra en a[], y el número inicial de tramos ficticios, d[].

En cada nivel (se corresponde con cada pasada del algoritmo), el número de tramos que se mezclan es el mínimo de los elementos de a[], siempre se encuentra en a[m-1], siendo m el número de archivos. Al escribir tramos mezclados en el archivo de salida, si ocurre que todos

los tramos son ficticios, la posición que le corresponde al archivo de salida en el *array* de tramos ficticios, `d[]`, se incrementa en 1.

En el algoritmo de mezcla que a continuación se escribe, `c[]` es una tabla de correspondencia para acceder a los `m-1` archivos, de tal forma que en `c[]` se encuentran los índices de los archivos que se están mezclando. Como puede haber archivos no activos debido a los tramos ficticios, `cd[]` es la tabla de correspondencia con los índices de los archivos activos, `k` es el número de archivos activos y `actv[]`, *array* lógico indexado por `c[i]`, está a `true` si el archivo correspondiente está activo. Las rotaciones de los archivos se realizan al finalizar un archivo de entrada durante la mezcla de tramos, se hacen moviendo los elementos de la tabla `c[]`. La mezcla de tramos se realiza de igual forma que en la mezcla múltiple: una función de lectura de ítems de todos los archivos y la selección del mínimo.

Algoritmo

```
desde i <-1 hasta m-1 hacer
   c[i] <- i
fin _ desde

repetir

   { Mezcla de tramos de los archivos c[1]...c[m-1] }

   z <- a[m-1] {número de tramos a mezclar en esta pasada}
   d[m] <- 0 {inicializa tramos ficticios en archivo de salida}
   <Preparar para escribir F[c[m]]>
   repetir
         { Mezcla de un tramo de los m-1 archivos }
      k <- 0  { número de ficheros activos}
      desde i <- 1 hasta m-1 hacer
         si d[i] > 0 entonces {es un tramo ficticio}
            d[i] <- d[i]-1
         sino {es un tramo real}
            k <- k+1
            cd[k] <- c[i]
         fin _ si
      fin _ desde

      si k > 0 entonces
         <Mezclar los tramos de los k archivos>
      sino {Todos los tramos son ficticios}
         d[m] <- d[m]+1 {tramo ficticio en el archivo destino}
      fin _ si
      z <- z-1 {tramo ya mezclado}
      hasta z = 0
         { Se han mezclado todos los tramos, el archivo de salida pasa a
         ser de entrada }
      <Preparar para lectura F[c[m]]>
      <Rotar los archivos en la tabla de correspondencia c[]>
      <Calcular los números de fibonacci, a[i], del siguiente nivel>
      Nivel <- Nivel-1
   hasta Nivel = 0
Fin
```

7.7.4. Mezcla polifásica *versus* mezcla múltiple

Las principales diferencias de la ordenación polifásica respecto a la mezcla equilibrada múltiple son:

1. En cada pasada hay un sólo archivo destino (salida), en vez de los `m/2` que necesita la mezcla equilibrada múltiple.
2. La finalidad de los archivos *(entrada, salida)* cambia en cada pasada, rotando los índices de los archivos. Esto se controla mediante un tabla de correspondencia de índices de archivo. En la mezcla múltiple siempre se intercambian `m/2` archivos origen (entrada) por `m/2` archivos destino (salida).
3. El número de archivos origen (de entrada) varía dependiendo del número de tramo en proceso. Éste se determina en el momento de empezar el proceso de mezcla de un tramo, a partir del contador `di` de tramos ficticios para cada archivo `i`. Puede ocurrir que `di >0` para todos los valores de `i, i=1..m-1`, lo que significa que hay se mezclan `m-1` tramos ficticios, dando lugar a un tramo ficticio en el archivo destino, a la vez se incrementará el elemento `d[]` correspondiente al archivo destino (de salida). Normalmente, se mezclarán tramos reales de cada archivo de entrada (se cumple que `di == 0`).
4. Ahora, el criterio de terminación de una fase radica en el número de tramos a ser mezclados en cada archivo. Puede ocurrir que se alcance el último registro del archivo `m-1` y sean necesarias más mezclas que utilicen tramos ficticios de ese archivo. En la fase de distribución inicial fueron calculados los números de Fibonacci de cada nivel de forma progresiva, hasta alcanzar el nivel en el que se agotó el archivo de entrada; ahora, partiendo de los números del último nivel, pueden recalcularse hacia atrás.

RESUMEN

La ordenación de archivos se denomina ordenación externa porque los registros no se encuentran en arrays (memoria interna), sino en dispositivos de almacenamiento masivo, como son los cartuchos, cd, discos duros...; por lo que se requieren algoritmos apropiados. Una manera trivial de realizar la ordenación de un archivo secuencial consiste en copiar los registros a otro archivo de acceso directo, o bien secuencial indexado, usando como clave el campo por el que se desea ordenar.

Si se desea realizar la ordenación de archivos utilizando solamente como estructura de almacenamiento auxiliar otros archivos secuenciales de formato similar al que se desea ordenar, hay que trabajar usando el esquema de *separación* y *mezcla*.

En el caso del algoritmo de *mezcla simple,* se opera con tres archivos análogos: el original y dos archivos auxiliares. El proceso consiste en recorrer el archivo original y copiar secuencias de sucesivos registros, alternativamente, en cada uno de los archivos auxiliares. A continuación, se mezclan las secuencias de los archivos y se copia la secuencia resultante en el archivo original. El proceso continúa de tal forma que, en cada pasada, la longitud de la secuencia es el doble de la longitud de la pasada anterior. Todo empieza con secuencias de longitud 1 y termina cuando se alcanza una secuencia de longitud igual al número de registros.

El algoritmo de *mezcla natural* también opera con tres archivos, pero se diferencia de la mezcla directa en que distribuye secuencias ordenadas en vez de secuencias de longitud fija.

Se estudian métodos avanzados de ordenación externa, como la *mezcla equilibrada múltiple* y la *mezcla polifásica*. En el primero se utiliza un número par de archivos auxiliares, la mitad de ellos son archivos de entrada y la otra mitad archivos de salida. El segundo se caracteriza por hacer una distribución inicial de tramos (según las secuencias de números de Fibonacci), para después realizar una *mezcla continuada* hasta obtener un único tramo ordenado.

La primera parte del capítulo revisa la jerarquía de clases para procesar archivos. Java contiene un paquete especializado en la entrada y salida de datos, el paquete java.io. Contiene un conjunto de clases organizadas jerárquicamente para tratar cualquier tipo de flujo de entrada o de salida de datos. Es necesaria la sentencia `import java.io.*` en los programas que utilicen objetos de alguna de estas clases. Todo se basa en la *abstracción* de flujo: *corriente* de bytes que entran o que salen de un dispositivo. Para Java, un archivo, cualquier archivo, es un flujo de bytes, e incorpora clases que procesan los flujos a bajo nivel, como secuencias de bytes. Para estructurar los bytes y formar datos a más alto nivel hay clases de más alto nivel; con estas clases se pueden escribir o leer directamente datos de cualquier tipo simple (entero, char...). Estas clases se enlazan con las clases de bajo nivel que, a su vez, se asocian a los archivos.

EJERCICIOS

7.1. Escribir las sentencias necesarias para abrir un archivo de caracteres, cuyo nombre y acceso se introduce por teclado, en modo lectura; en el caso de que el resultado de la operación sea erróneo, abrir el archivo en modo escritura.

7.2. Un archivo contiene enteros positivos y negativos. Escribir un método para leer el archivo y determinar el número de enteros negativos.

7.3. Escribir un método para copiar un archivo. El método tendrá dos argumentos de tipo cadena, el primero es el archivo original y el segundo es el archivo destino. Utilizar flujos `FileInputStream` y `FileOutputStream`.

7.4. Una aplicación instancia objetos de las clases *NumeroComplejo* y *NumeroRacional*. La primera tiene dos variables instancia de tipo `float`, *parteReal* y *parteImaginaria*. La segunda clase tiene definidas tres variables *numerador* y *denominador,* de tipo `int`, y *frac,* de tipo `double`. Escribir la aplicación de tal forma que los objetos sean persistentes.

7.5. El archivo F, almacena registros con un campo clave de tipo entero. Suponer que la secuencia de claves que se encuentra en el archivo es la siguiente:

```
14 27 33 5 8 11 23 44 22 31 46 7 8 11 1 99 23 40 6 11 14 17
```

Aplicando el algoritmo de mezcla directa, realizar la ordenación del archivo y determinar el número de pasadas necesarias.

7.6. Considerando el mismo archivo que el del Ejercicio 7.5, aplicar el algoritmo de mezcla natural para ordenar el archivo. Comparar el número de pasadas con las obtenidas en el ejercicio anterior.

7.7. Un archivo secuencial `F` contiene registros y quiere ser ordenado utilizando 4 archivos auxiliares. Suponiendo que la ordenación se desea hacer respecto a un campo de tipo entero, con estos valores:

```
22 11 3 4 11 55 2 98 11 21 4 3 8 12 41 21 42 58 26 19 11 59 37 28 61 72 47
```

aplicar el algoritmo de mezcla equilibrada múltiple y obtener el número de pasadas necesarias para su ordenación.

7.8. Con el mismo archivo que el del Ejercicio 7.7, y también con `m = 4` archivos auxiliares, aplicar el algoritmo de mezcla polifásica. Comparar el número de pasadas realizadas en ambos métodos.

PROBLEMAS

7.1. Escribir un programa que compare dos archivos de texto (caracteres). El programa ha de mostrar las diferencias entre el primer archivo y el segundo, precedidas del número de línea y de columna.

7.2. Un atleta utiliza un pulsómetro para sus entrenamientos, que almacena sus pulsaciones cada 15 segundos, durante un tiempo máximo de 2 horas. Escribir un programa para almacenar en un archivo los datos del pulsómetro del atleta, de tal forma que el primer registro contenga la fecha, hora y tiempo en minutos de entrenamiento y, a continuación, los datos del pulsómetro por parejas: tiempo, pulsaciones.

7.3. Las pruebas de acceso a la Universidad Pontificia de Guadalajara, UPGA, constan de 4 apartados, cada uno de los cuales se puntúa de 1 a 25 puntos. Escribir un programa para almacenar en un archivo los resultados de las pruebas realizadas, de tal forma que se escriban objetos con los siguientes datos: nombre del alumno, puntuación de cada apartado y puntuación total.

7.4. Dado el archivo de puntuaciones generado en el Problema 7.3, escribir un programa para ordenar el archivo utilizando el método de ordenación externa *de mezcla natural*.

7.5. Supóngase que se dispone del archivo ordenado de puntuaciones de la UPGA (Problema 7.4) y del archivo de la Universidad de Salamanca que consta de objetos con los mismos datos y también está ordenado. Escribir un programa que mezcle ordenadamente los dos archivos en un tercero.

7.6. Se tiene guardado en un archivo a los habitantes de la comarca de Pinilla. Los datos de cada persona son los siguientes: primer apellido (campo clave), segundo apellido (campo secundario), edad, años de estancia y estado civil. Escribir un programa para ordenar el archivo por el método de mezcla equilibrada múltiple. Utilizar 6 archivos auxiliares.

7.7. Una farmacia desea mantener su *stock* de medicamentos en una archivo. De cada producto, interesa guardar el código, el precio y la descripción. Escribir un programa que genere el archivo pedido almacenando los objetos de manera secuencial.

7.8. Escribir un programa para ordenar el archivo que se ha generado en el Problema 7.7. Utilizar el método de ordenación de mezcla polifásica, con `m = 5` archivos auxiliares.

7.9. Implementar un método de ordenación externa, suponiendo un archivo de números enteros, con dos archivos auxiliares. La separación inicial del archivo en tramos sigue la siguiente estrategia: se leen 20 elementos del archivo en un *array* y se ordenan con el método de ordenación interna *Quicksort*. A continuación, se escribe el elemento menor del *array* en el archivo auxiliar y se lee el siguiente elemento del archivo origen. Si el elemento leído es mayor que el elemento escrito (forma parte del tramo actual), entonces se inserta en orden en el *subarray* ordenado; en caso contrario, se añade en las posiciones libres del *array*. Debido a que los elementos del *array* se extraen por la cabeza, quedan posiciones libres por su extremo inferior. El tramo termina en el momento en que el *subarray* ordenado queda vacío. Para formar el siguiente tramo, se empieza ordenando el *array* (recordar que los elementos que no formaban parte del tramo se iban añadiendo al *array*) y, después, el proceso continúa de la misma forma: escribir el elemento menor en otro archivo auxiliar y leer elemento del archivo origen... Una vez que se realiza la distribución la fase de mezcla es igual que en los algoritmos de mezcla directa o natural.





CAPITULO 8
Listas enlazadas

Objetivos

Con el estudio de este capítulo, usted podrá:

- Distinguir entre una estructura secuencial y una estructura enlazada.
- Definir el tipo abstracto de datos *Lista*.
- Conocer las operaciones básicas de la listas enlazadas.
- Implementar una lista enlazada para un tipo de elemento.
- Aplicar la estructura *Lista* para almacenar datos en el desarrollo de aplicaciones.
- Definir una lista doblemente enlazada.
- Definir una lista circular.
- Implementar listas enlazadas ordenadas.
- Realizar una lista genérica.

Contenido

Conceptos clave

- Enlace.
- Estructura enlazada.
- Lista circular.
- Lista doble.
- Matriz dispersa.
- Nodo.
- Recorrer una lista.
- Tipo abstracto de dato (TAD).

INTRODUCCIÓN

En este capítulo se comienza el estudio de las estructuras de datos dinámicas. Al contrario que las estructuras de datos estáticas (*arrays* —listas, vectores y tablas— y *estructuras*), cuyo tamaño en memoria permanece inalterable durante la ejecución del programa, las estructuras de datos dinámicas crecen y se contraen a medida que se ejecuta el programa.

La estructura de datos que se estudiará en este capítulo es la *lista enlazada* (ligada o encadenada, *linked list*): una colección de elementos (denominados nodos) dispuestos uno a continuación de otro, cada uno de ellos conectado al siguiente por un "enlace" o "referencia". En el capítulo se desarrollan métodos para insertar, buscar y borrar elementos en listas enlazadas. De igual modo, se muestra el tipo abstracto de datos *(TAD)* que representa las listas enlazadas.

Las listas enlazadas son estructuras muy flexibles y con numerosas aplicaciones en el mundo de la programación

8.1. FUNDAMENTOS TEÓRICOS DE LISTAS ENLAZADAS

Las estructuras de datos lineales de elementos homogéneos (listas, tablas, vectores) utilizaban *arrays* para implementar tales estructuras, siendo los elementos de tipo primitivo `(int, long, double...);` también se ha utilizado la clase `Vector`, aunque los elementos, en este caso, han de ser referencias. Esta técnica obliga a fijar por adelantado el espacio a ocupar en memoria, de modo que, cuando se desea añadir un nuevo elemento que rebase el tamaño prefijado del *array,* no es posible realizar la operación sin que se produzca un error en tiempo de ejecución. Esta característica se debe a que los *arrays* hacen un uso ineficiente de la memoria. Gracias a la asignación dinámica de variables, se pueden implementar listas de modo que la memoria física utilizada se corresponda con el número de elementos de la tabla; para ello, se recurre a las `referencias` *(apuntadores)* que hacen un uso más eficiente de la memoria, como ya se ha visto con anterioridad.

Una **lista enlazada** es una colección o secuencia de elementos dispuestos uno detrás de otro, en la que cada elemento se conecta al siguiente elemento por un "enlace" o "referencia". La idea básica consiste en construir una lista cuyos elementos, llamados **nodos,** se componen de dos partes *(campos)*: la primera parte contiene la información y es, por consiguiente, un valor de un tipo genérico (denominado `Dato, TipoElemento, Info`, etc.), y la segunda parte es una referencia (denominado `enlace` o `sgte`) que apunta (enlaza) al siguiente elemento de la lista.

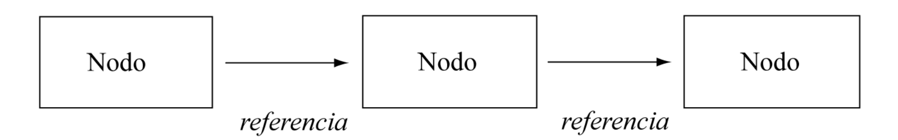
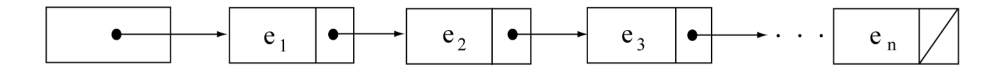


Figura 8.1 Lista enlazada (representación simple)

La representación gráfica más extendida es aquella que utiliza una caja (un rectángulo) con dos secciones en su interior. En la primera sección se escribe el elemento o valor del dato, y en la segunda sección, el enlace o referencia mediante una flecha que sale de la caja y apunta al nodo siguiente.



e_1, e_2, ...e_n, en son valores del tipo `TipoElemento`

Figura 8.2 Lista enlazada (representación gráfica típica)

Para recordar

Una **lista enlazada** consta de un número de elementos, y cada elemento tiene dos componentes *(campos)*, una referencia al siguiente elemento de la lista y un valor, que puede ser de cualquier tipo

Los enlaces se representan por flechas para facilitar la comprensión de la conexión entre dos nodos e indicar que el enlace tiene la dirección en memoria del siguiente nodo. Los enlaces también sitúan los nodos en una secuencia. En la Figura 8.2, los nodos forman una secuencia desde el primer elemento (e_1) al último elemento (e_n). El primer nodo se enlaza al segundo, éste se enlaza al tercero, y así sucesivamente hasta llegar al último nodo, que debe ser representado de forma diferente para significar que este nodo que no se enlaza a ningún otro. La Figura 8.3 muestra diferentes representaciones gráficas utilizadas para dibujar el campo enlace del último nodo.

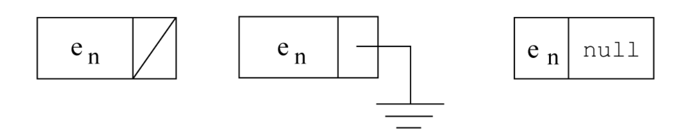


Figura 8.3 Diferentes representaciones gráficas del nodo último

8.2. CLASIFICACIÓN DE LISTAS ENLAZADAS

Las listas se pueden dividir en cuatro categorías :

- *Listas simplemente enlazadas*. Cada nodo (elemento) contiene un único enlace que lo conecta al nodo siguiente o nodo sucesor. La lista es eficiente en recorridos directos ("adelante").
- *Listas doblemente enlazadas*. Cada nodo contiene dos enlaces, uno a su nodo predecesor y otro a su nodo sucesor. La lista es eficiente tanto en recorrido directo ("adelante") como en recorrido inverso ("atrás").
- *Lista circular simplemente enlazada*. Una lista enlazada simplemente en la que el último elemento (cola) se enlaza al primer elemento (cabeza) de tal modo que la lista puede ser recorrida de modo circular ("en anillo").
- *Lista circular doblemente enlazada*. Una lista doblemente enlazada en la que el último elemento se enlaza al primer elemento y viceversa. Esta lista se puede recorrer de modo circular ("en anillo") tanto en dirección directa ("adelante") como inversa ("atrás").

La implementación de cada uno de los cuatro tipos de estructuras de listas se puede desarrollar utilizando referencias.

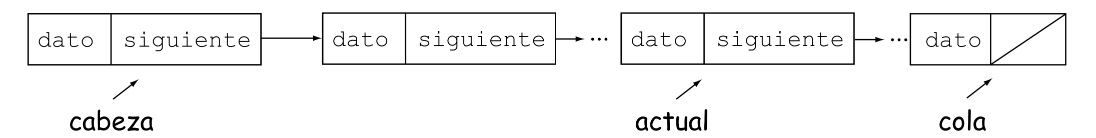


Figura 8.4 Representación gráfica de una lista enlazada

El primer nodo, **frente,** de una lista es el nodo apuntado por **cabeza.** La lista encadena nodos juntos desde el frente al final (**cola**) de la lista. El final se identifica como el nodo cuyo campo referencia tiene el valor `null`. La lista se recorre desde el primero al último nodo; en cualquier punto del recorrido la posición *actual* se referencia por el puntero *(pointer)* `actual`. Una lista vacía (no contiene nodos), se representa con el puntero `cabeza` con nulo `(null)`.

8.3. TIPO ABSTRACTO DE DATOS (TAD) LISTA

Una lista se utiliza para almacenar información del mismo tipo, con la característica de que puede contener un número indeterminado de elementos y que estos elementos mantienen un orden explícito. Este ordenamiento explícito implica que cada elemento (un nodo de la lista) contiene la dirección del siguiente elemento.

Una lista es una estructura de datos dinámica. El número de nodos puede variar rápidamente en un proceso, aumentando por inserciones o disminuyendo por eliminación de nodos.

Las inserciones se pueden realizar por cualquier punto de la lista: por la cabeza (inicio), por el final, a partir o antes de un nodo determinado de la lista. Las eliminaciones también se pueden realizar en cualquier punto; además, se eliminan nodos dependiendo del campo de información o dato que se desea suprimir de la lista.

8.3.1. Especificación formal del TAD Lista

Matemáticamente, *una lista* es una secuencia de cero o más elementos de un determinado tipo.

$(a_1, a_2, a_3, \ldots, a_n)$ donde $n >= 0$,
si $n = 0$ la lista es vacía.

Los elementos de la lista tienen la propiedad de que sus elementos están ordenados de forma lineal, según las posiciones que ocupan en la misma. Se dice que a_i precede a a_{i+1} para $i = 1 \ldots, n-1$; y que a_i sucede a a_{i-1} para $i = 2 \ldots n$.

Para formalizar el *tipo de dato abstracto* `Lista` a partir de la noción matemática, se define un conjunto de operaciones básicas con objetos de tipo `Lista`. Las operaciones son:

$\forall$ L $\in$ Lista, $\forall$ x $\in$ Lista, $\forall$ p $\in$ puntero

`Listavacia(L)`	Inicializa la lista `L` como lista vacía.
`Esvacia(L)`	Determina si la lista `L` está vacía.
`Insertar(L,x,p)`	Inserta en la lista `L` un nodo con el campo dato `x`, delante del nodo de dirección `p`.

`Localizar(L,x)`	Devuelve la posición/dirección donde está el campo de información `x`.
`Suprimir(L,x)`	Elimina de la lista el nodo que contiene el dato `x`.
`Anterior(L,p)`	Devuelve la posición/dirección del nodo anterior a `p`.
`Primero(L)`	Devuelve la posición/dirección del primer nodo de la lista `L`.
`Anula(L)`	Vacía la lista `L`.

Estas operaciones son las que pueden considerarse básicas para manejar listas. En realidad, la decisión de qué operaciones son las básicas depende de las características de la aplicación que se va a realizar con los datos de la lista. También dependerá del tipo de representación elegido para las listas. Así, para añadir nuevos nodos a una lista, se implementan, además de `insertar()`, versiones de ésta como:

`inserPrimero(L,x)`	Inserta un nodo con el dato `x` como primer nodo de la lista `L`.
`inserFinal(L,x)`	Inserta un nodo con el dato `x` como último nodo de la lista `L`.

Una operación típica de toda estructura de datos enlazada es `recorrer`. Consiste en visitar cada uno de los datos o nodos de que consta. En las listas enlazadas, esta operación se realiza normalmente desde el nodo *cabeza* al último nodo o *cola* de la lista.

8.4. OPERACIONES EN LISTAS ENLAZADAS

La implementación del TAD `Lista` requiere, en primer lugar, declarar la clase `Nodo`, en la que se combinarán sus dos partes: el *dato* (entero, real, doble, carácter o referencias a objetos) y un *enlace*. Además, la clase `Lista` con las operaciones y el atributo con la cabeza de la lista. Las operaciones tendrán las siguientes funciones:

- *Inicialización o creación*.
- *Insertar elementos en la lista.*
- *Eliminar elementos de la lista.*
- *Buscar elementos de la lista*.
- *Recorrer la lista enlazada*.
- *Comprobar si la lista está vacía.*

8.4.1. Declaración de un nodo

Una lista enlazada se compone de una serie de nodos enlazados mediante referencias. En Java, se declara una clase para contener las dos partes del nodo: `dato` y `enlace`. Por ejemplo, para una lista enlazada de números enteros, la clase `Nodo` es:

```
class Nodo
{
   int dato;
   Nodo enlace;
   public Nodo(int t)
   {
      dato = t;
      enlace = null;
   }
}
```

En la declaración, la visibilidad de los dos campos del nodo, dato y enlace, es la del `package` (visibilidad por defecto). De esa manera, los métodos que implementan las operaciones de listas pueden acceder a ellos al estar en el mismo paquete. El constructor inicializa el objeto `Nodo` a una dato y el enlace a la referencia `null`.

Dado que los tipos de datos que se pueden incluir en una lista pueden ser de cualquier tipo (enteros, dobles, caracteres o cualquier objeto) con el objetivo de que el tipo de dato de cada nodo se pueda cambiar con facilidad, se define a veces la clase `Elemento` como una generalización del tipo de dato. En ese caso, se utiliza una referencia a `Elemento` dentro del nodo, como se muestra a continuación:

```
class Elemento
{
   // ...
}

class Nodo
{
   Elemento dato;
   Nodo enlace;
}
```

Entonces, si se necesita cambiar el tipo de elemento en los nodos, sólo tendrá que cambiar el tipo encerrado en la clase `Elemento`. Siempre que un método necesite referirse al tipo del dato del nodo puede utilizar el nombre `Elemento`.

Otra manera de representar el tipo de los datos de una lista es definiéndolo como del tipo `Object` (superclase base de todas las clases).

Ejemplo 8.1

La clase `Punto`*, representa un punto en el plano de coordenadas (x, y). La clase* `Nodo` *con un campo dato referencia a objetos de la clase* `Punto`*. Estas clases formarán parte del paquete* `ListaPuntos`.

```
package ListaPuntos;

public class Punto
{
  double x, y;

   public Punto(double x, double y)
   {
      this.x = x;
      this.y = y;
   }
   public Punto()  // constructor por defecto
   {
      x = y = 0.0;
   }
}
```

La clase `Nodo` que se escribe a continuación tiene como campo `dato` una referencia a `Punto` y como campo enlace una referencia a otro `Nodo`. Se definen dos constructores: el

primero inicializa `dato`, un objeto `Punto` y `enlace` a `null`; el segundo inicializa `enlace` de tal forma que referencia un `Nodo`.

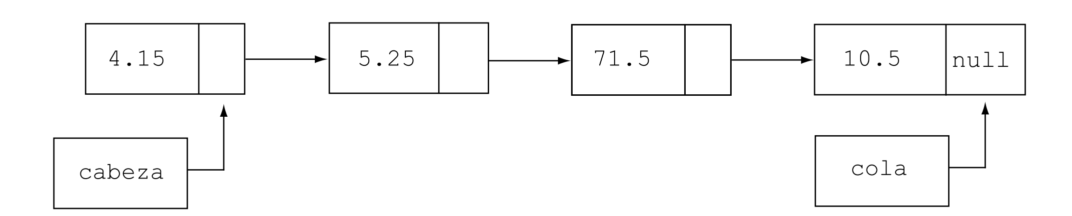
```
package ListaPuntos;

public class Nodo
{
   Punto dato;
   Nodo enlace;

   public Nodo(Punto p)
   {
      dato = p;
      enlace = null;
   }
   public Nodo(Punto p, Nodo n)
   {
      dato = p;
      enlace = n;
   }
}
```

8.4.2. Acceso a la lista: cabecera y cola

Cuando se construye y se utiliza una lista enlazada en una aplicación, el acceso a la lista se hace mediante una o más *referencias* a los nodos. Normalmente, se accede a partir del primer nodo de la lista, llamado **cabeza** o **cabecera** de la lista. Una referencia al primer nodo se llama referencia **cabeza.** En ocasiones, se mantiene también una referencia al último nodo de la lista enlazada. El último nodo es la **cola** de la lista, y una referencia al último nodo es la referencia **cola.**



Definición nodo

```
class Nodo
{
  double dato;
  Nodo enlace;
  public Nodo(){;}
}
```

Definición de referencias

```
Nodo cabeza;

Nodo cola;
```

Figura 8.5 Declaraciones de tipos en lista enlazada

Cada referencia a un nodo debe ser declarada como una variable referencia. Por ejemplo, si se mantiene una lista enlazada con una referencia de cabecera y otra de cola, se deben declarar dos variables referencia:

```
Nodo cabeza;
Nodo cola;
```

A recordar

La construcción y manipulación de una lista enlazada requiere el acceso a los nodos de la lista a través de una o más referencias a nodos. Normalmente, se incluye una referencia al primer nodo *(cabeza)* y además, en algunas aplicaciones, una referencia al último nodo *(cola)*.

La Figura 8.6 muestra una lista a la que se accede con la referencia `cabeza`; cada nodo está enlazado con el siguiente nodo. El último nodo, cola o final de la lista, no se enlaza con otro nodo, por lo que su campo de enlace contiene la *referencia nulo,* `null`. La palabra `null` representa la **referencia nulo,** que es una constante especial de Java. Se puede utilizar `null` para cualquier valor de referencia que no apunte a objeto alguno. La referencia `null` se utiliza, normalmente, en dos situaciones:

- En el campo *enlace* del último nodo (final o cola) de una lista enlazada.
- Como valor de la referencia `cabeza` para una lista enlazada que no tiene nodos. Tal lista se denomina **lista vacía (cabeza = null)**.

La referencia `null` se puede asignar a una variable referencia con una sentencia de asignación ordinaria. Por ejemplo:

```
Nodo cabeza;
cabeza = null;
```

```
[   •]──→[4.15|•]──→[5.25|•]──→[71.5|•]──→[10.5|null]
cabeza
```

Figura 8.6 Referencia `null`

Nota de programación

La referencia cabeza (y cola) de una lista enlazada, normalmente, se inicializa a `null`, indica lista vacía (no tiene nodos), cuando se inicia la construcción de una lista. Cualquier método que se escriba para implementar listas enlazadas debe poder manejar un referencia de cabeza (y de cola) `null`.

Error

Uno de los errores típicos en el tratamiento de referencias consiste en escribir la expresión *p.miembro* cuando el valor de la referencia `p` es `null`.

8.4.3. Construcción de una lista

La creación de una lista enlazada entraña la definición de, al menos, la clases `Nodo` y `Lista`. Opcionalmente, se declara la clase `Elemento` que define las características de cada dato del nodo.

La clase `Lista` define el atributo `cabeza` o primero, referencia a `Nodo`, para acceder a los elementos de la lista. Normalmente, no es necesario definir el atributo referencia `cola`. El constructor de `Lista` inicializa `primero` a `null` *(lista vacía).*

Los métodos de la clase `Lista` implementan las operaciones de una lista enlazada: *inserción*, *búsqueda*... Además, el método `crearLista()`, construye iterativamente el primer elemento `(primero)` y los elementos sucesivos de una lista enlazada.

Ejemplo 8.2

Crear una lista enlazada de elementos que almacenen datos de tipo entero.

La declaración de la clase `Nodo` es:

```
package ListaEnteros;

public class Nodo
{
   int dato;
   Nodo enlace;

   public Nodo(int x)
   {
      dato = x;
      enlace = null;
   }
   public Nodo(int x, Nodo n)
   {
      dato = x;
      enlace = n;
   }
   public int getDato()
   {
      return dato;
   }
   public Nodo getEnlace()
   {
      return enlace;
   }
   public void setEnlace(Nodo enlace)
   {
      this.enlace = enlace;
   }

}
```

El segundo constructor de `Nodo` enlaza el nodo creado con otro. El siguiente paso para construir la lista es declarar la clase `Lista`:

```
package ListaEnteros;

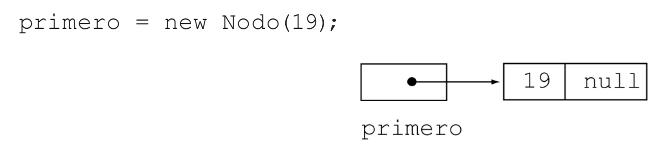
public class Lista
{
```

```
      private Nodo primero;

      public Lista()
      {
         primero = null;
   }
   //...
```

La referencia `primero` (también se puede llamar `cabeza`) se ha inicializado en el constructor a un valor *nulo,* es decir, a *lista vacía*. A continuación, se muestra cúal será el comportamiento del método `crearLista()`. En primer lugar, se crea un nodo con un valor y su referencia se asigna a `primero`:

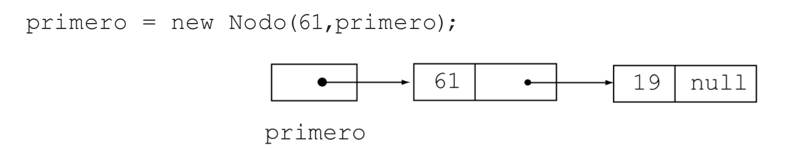
```
primero = new Nodo(19);
```



La referencia `primero` apunta al primer nodo y al campo dato se le ha dado un valor cualquiera, `19`. El `enlace` del nodo toma el valor *nulo,* al no existir un nodo siguiente.

La operación de crear un nodo se puede realizar en un método al que se pasa el valor del campo `dato` y del campo `enlace`. Si ahora se desea añadir un nuevo elemento con el valor `61` y situarlo en el primer lugar de la lista, se escribe simplemente:

```
primero = new Nodo(61,primero);
```



Por último, para obtener una lista compuesta de `4`, `61`, `19` habría que ejecutar:

```
primero = new Nodo(4,primero);
```

A continuación, se escribe el método `crearLista()` que codifica las acciones descritas anteriormente. El método crea una lista iterativamente hasta leer el valor clave `-1`. Los valores de los nodos se leen del teclado, con llamadas al método `leerEntero()`. El método `crearLista()` devuelve una referencia al objeto lista creado `(this)`.

```
private int leerEntero() {;}

public Lista crearLista()
{
   int x;
   primero = null;
   do {
      x = leerEntero();
      if (x != -1)
```

```
        {
           primero = new Nodo(x,primero);
        }
     }while  (x != -1);
     return this;
   }
```

8.5. INSERCIÓN DE UN ELEMENTO EN UNA LISTA

El nuevo elemento que se desea incorporar a una lista se puede insertar de distintas formas, según la posición o punto de inserción:

- En la cabeza de la lista (elemento primero).
- En el final de la lista (elemento último).
- Antes de un elemento especificado.
- Después de un elemento especificado

8.5.1. Insertar un nuevo elemento en la cabeza de la lista

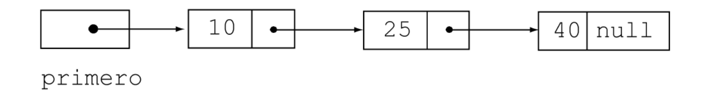
La posición más facil y, a la vez, más eficiente en donde insertar un nuevo elemento de una lista es en la *cabeza*, es decir, por el primer nodo de la lista. El proceso de inserción se resume en este algoritmo:

1. Crear un nodo e inicializar el campo `dato` al nuevo elemento. La referencia del nodo creado se asigna a `nuevo`, variable local del método.
2. Hacer que el campo `enlace` del nuevo nodo apunte a la *cabeza* `(primero)` de la lista original.
3. Hacer que `primero` apunte al nodo que se ha creado.

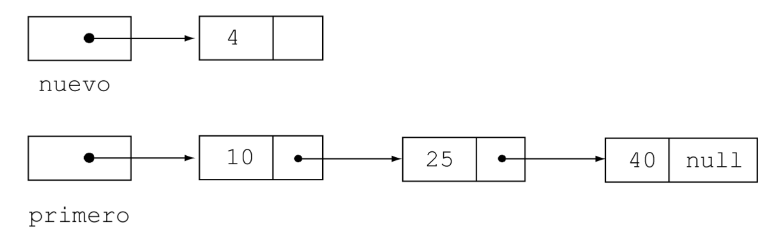
El Ejemplo 8.3 inserta un elemento por la *cabeza* de una lista siguiendo los pasos del algoritmo. A continuación, se escribe el código Java.

Ejemplo 8.3

Una lista enlazada contiene tres elementos, 10, 25 y 40. Insertar un nuevo elemento, 4, en cabeza de la lista.



Paso 1



Código Java

```
Nodo nuevo;
nuevo = new Nodo(entrada); // asigna un nuevo nodo
```

Paso 2

El campo `enlace` del nuevo nodo apunta al nodo `primero` actual de la lista.

Código Java

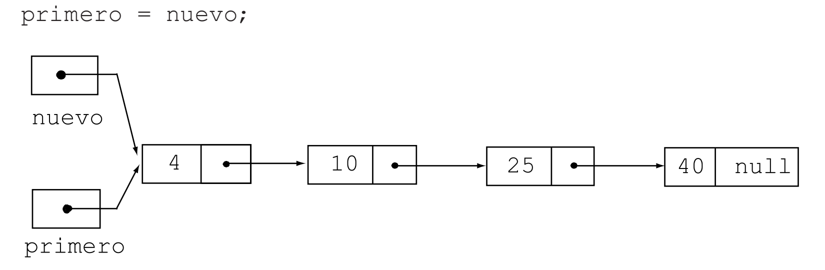
```
nuevo.enlace = primero
```

Paso 3

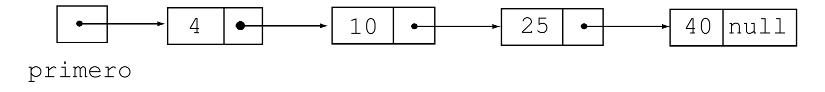
Se cambia la referencia de `primero` para que apunte al nodo creado; es decir, `primero` apunta al mismo nodo al que apunta `nuevo`.

Código Java

```
primero = nuevo;
```



En este momento, el método de insertar termina su ejecución, la variable local `nuevo` desaparece y sólo permanece la referencia al primer nodo de la lista: `primero`.



El código fuente del método `insertarCabezaLista`:

```
public Lista insertarCabezaLista(Elemento entrada)
{
   Nodo nuevo ;
   nuevo = new Nodo(entrada);
   nuevo.enlace = primero;  // enlaza nuevo nodo al frente de la lista
   primero= nuevo;          // mueve primero y apunta al nuevo nodo
   return this;             // devuelve referencia del objeto Lista
}
```

Caso particular

El método `insertarCabezaLista` también actúa correctamente si se trata de añadir un primer nodo o elemento a una lista vacía. En este caso, como ya se ha comentado, `primero` apunta a `null` y termina apuntando al nuevo nodo de la lista enlazada.

Ejercicio 8.1

Crear una lista de números aleatorios. Insertar los nuevos nodos por la cabeza de la lista. Un vez creada la lista, se han de recorrer los nodos para mostrar los número pares.

Se va a crear una lista enlazada de números enteros; para ello se definen la clase `Nodo` y la clase `Lista`. En esta última se definen los métodos `insertarCabezaLista()` que añade un nodo a la lista, siempre como nodo cabeza; y el método `visualizar()` que recorre la lista escribiendo el campo dato de cada nodo. Desde el método `main()` se crea un objeto `Lista`, se llama iterativamente al método que añade nuevos elementos, y por último se llama a `visualizar()` para mostrar los elementos. Para generar números aleatorios se utiliza la clase `Random` (paquete `java.util`).

El programa consta del paquete `ListaEnteros`, con la clase `Nodo` y la clase `Lista`, y la clase principal con el método `main`.

```
package ListaEnteros;
// clase Nodo con las dos partes de un nodo y su constructor
public class Nodo
{
   int dato;
   Nodo enlace;

   public Nodo(int x)
   {
      dato = x;
      enlace = null;
   }
   public int getDato()
   {
      return dato;
   }
   public Nodo getEnlace()
   {
      return enlace;
   }
   public void setEnlace(Nodo enlace)
   {
      this.enlace = enlace;
   }

}
   /* clase Lista con las operaciones: insertar por la cabeza y
      visualizar (recorre los nodos) para mostrar los datos. Además,
      el atributo primero, que apunta al primer nodo.
   */
package ListaEnteros;
public class Lista
```

```
{
   protected Nodo primero;

   public Lista()
   {
      primero = null;
   }
   public Lista insertarCabezaLista(int entrada)
   {
      Nodo nuevo ;
      nuevo = new Nodo(entrada);
      nuevo.enlace = primero;
      primero = nuevo;
      return this;
   }
   public void visualizar()
   {
      Nodo n;
      int k = 0;

      n = primero;
      while (n != null)
      {
         System.out.print(n.dato + "  ");
         n = n.enlace;
         k++;
         System.out.print( (k%15 != 0 ? " " : "\n"));
      }
   }
}
// clase con método main
import java.util.*;
import ListaEnteros.*;

public class ListaAleatoria
{
   public static void main(String [] a)
   {
      Random r;
      int d;
      Lista lista;
      int k;

      r = new Random();
      lista = new Lista();           // crea lista vacía
      k = Math.abs(r.nextInt()% 55); // número de nodos
      // se insertan elementos en la lista
      for (; k > 0; k-- )
      {
         d = r.nextInt() % 99 ;
        lista.insertarCabezaLista(d);
      }
      // recorre la lista para escribir sus elementos
      System.out.println("Elementos de la lista generados al azar");
      lista.visualizar();
   }
}
```

8.5.2. Inserción al final de la lista

La inserción al final de la lista es menos eficiente debido a que, normalmente, no se tiene un puntero al último nodo y, entonces, se ha de seguir la traza desde la cabeza de la lista hasta el último nodo para, a continuación, realizar la inserción.

Una vez que la variable `ultimo` apunta al final de la lista, es decir, al último nodo, las sentencias siguientes insertan un nodo al final:

```
ultimo.enlace = new Nodo(entrada);
ultimo = ultimo.enlace;
```

La primera sentencia crea un nodo, inicializando su `dato` a `entrada`. El campo `enlace` del último nodo queda apuntando al nodo creado y así se enlaza, como nodo final, a la lista y la última sentencia pone la variable `ultimo` al nuevo último nodo de la lista. La operación es un método de la clase `Lista`.

```
public Lista insertarUltimo(Nodo ultimo, Elemento entrada)
{
   ultimo.enlace = new Nodo(entrada);
   ultimo = ultimo.enlace;
   return this;
}
```

8.5.3. Insertar entre dos nodos de la lista

La inserción de un nuevo nodo no se realiza siempre al principio (en cabeza) de la lista o al final, puede hacerse entre dos nodos cualesquiera de la lista. Por ejemplo, en la lista de la Figura 8.7, se quiere insertar el elemento `75` entre los nodos con datos `25` y `40`.

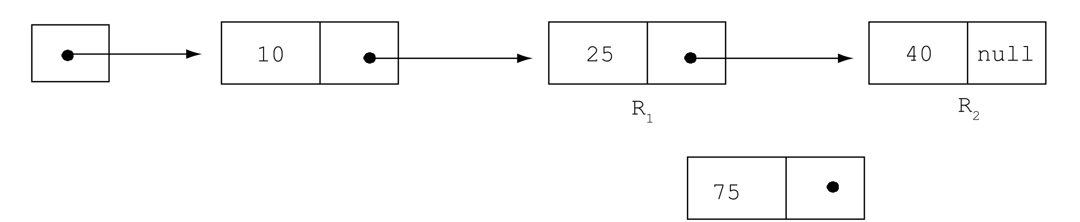


Figura 8.7 Inserción entre dos nodos

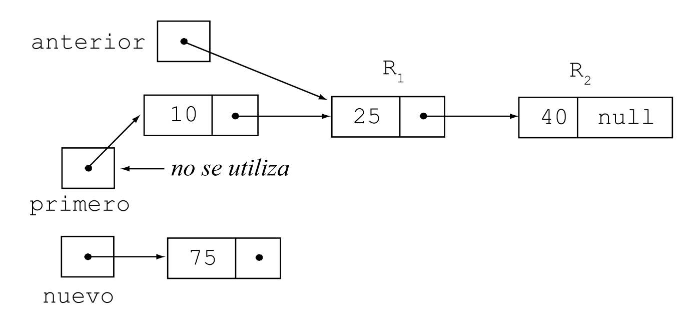
El algoritmo para la operación *insertar* entre dos nodos `(n1, n2)` requiere las siguientes etapas:

1. Crear un nodo con el nuevo elemento y el campo `enlace` a `null`. La referencia al nodo se asigna a `nuevo`.
2. Hacer que el campo `enlace` del nuevo nodo apunte al nodo `n2`, ya que el nodo creado se ubicará justo antes de `n2` (en la Figura 8.7, el nodo `40`).
3. La variable referencia `anterior` tiene la dirección del nodo `n1` (en la Figura 8.7, el nodo `25`), y eso exige hacer que `anterior.enlace` apunte al nodo creado.

A continuación se muestran gráficamente las etapas del algoritmo y el código que implementa la operación.

Etapa 1

Se crea un nodo con el dato `75`. La variable `anterior` apunta al nodo `n1`.

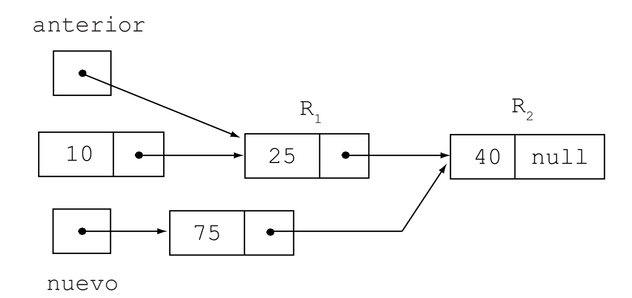


Código Java

```
nuevo = new Nodo(entrada);
```

Etapa 2

El campo enlace del nodo creado debe apuntar a `n2`. La dirección de `n2` se consigue con `anterior.enlace`:

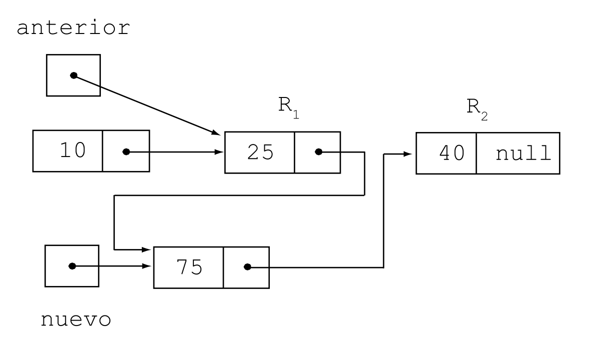


Código Java

```
nuevo.enlace = anterior.enlace
```

Etapa 3

Por último, el campo enlace del nodo `n1 (anterior)` debe apuntar al nodo creado:



Después de ejecutar todas las sentencias de las sucesivas etapas, la nueva lista comenzará en el nodo `10`, seguirán `25`, `75` y, por último, `40`.

La operación es un método de la clase `Lista`:

```
public Lista insertarLista(Nodo anterior, Elemento entrada)
{
   Nodo nuevo;
   nuevo = new Nodo(entrada);
   nuevo.enlace = anterior.enlace;
   anterior.enlace = nuevo;
   return this;
}
```

Antes de llamar al método `insertarLista()` es necesario buscar la dirección del nodo `n1`, esto es, del nodo a partir del cual se enlazará el nodo que se va a crear.

Una sobrecarga del método tiene como argumentos el dato a partir del cual se realiza el enlace, y el dato del nuevo nodo. El algoritmo de esta versión, busca primero el nodo con el dato *testigo,* a partir del cual se inserta y, a continuación, se realizan los mismos enlaces que en el método anterior.

Código Java

```
public Lista insertarLista(Elemento testigo, Elemento entrada)
{
   Nodo nuevo, anterior;

   anterior = buscarLista(testigo);
   if (anterior != null)
   {
      nuevo = new Nodo(entrada);
      nuevo.enlace = anterior.enlace;
      anterior.enlace = nuevo;
   }
   return this;
}
```

8.6. BÚSQUEDA EN LISTAS ENLAZADAS

La operación *búsqueda* de un elemento en una lista enlazada recorre la lista hasta encontrar el nodo con el elemento. El algoritmo que se utiliza para localizar un elemento en una lista enlazada, una vez encontrado el nodo, devuelve la referencia a ese nodo (en caso negativo, devuelve `null`). Otro planteamiento es que el método devuelve `true` si encuentra el nodo con el elemento y `false` si no está en la lista .

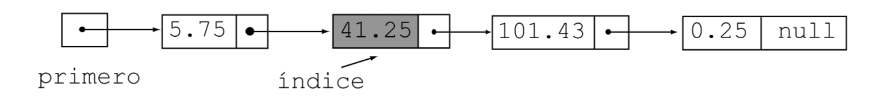


Figura 8.8 Búsqueda en una lista

El método `buscarLista` utiliza la referencia `indice` para recorrer la lista, nodo a nodo. El bucle de búsqueda inicializa `indice` al nodo *cabeza* `(primero)`, compara el nodo referenciado por `indice` con el elemento buscado si coincide la búsqueda, termina, en caso contrario; `indice` avanza al siguiente nodo. La búqueda termina cuando se encuentra el nodo, o bien cuando se ha recorrido la lista, y entonces `indice` toma el valor `null`. La búsqueda del elemento `41.25` en la lista de la Figura 8.8 termina en el segundo nodo, y devuelve la referencia, `indice,` a ese nodo.

La comparación entre el dato buscado y el dato del nodo que realiza el método `buscarLista()` utiliza el operador == por claridad; realmente, sólo se utiliza dicho operador si los datos son de tipo simple `(int, double...)`. Normalmente, los datos de los nodos son objetos, y entonces se utiliza el método `equals()` que compara dos objetos.

Código Java

```
public Nodo buscarLista(Elemento destino)
{
   Nodo indice;
   for (indice = primero; indice!= null; indice = indice.enlace)
      if (destino == indice.dato)   // (destino.equals(indice.dato))
         return indice;
   return null;
}
```

Ejemplo 8.4

Escribir un método de búsqueda alternativo para encontrar la dirección de un nodo dada su posición en una lista enlazada.

El método es un miembro público de la clase `Lista`; por consiguiente, accede a los miembros de la clase escribiendo simplemente su identificador. El nodo buscado se especifica por su posición en la lista; para ello, se considera posición `1` la correspondiente al nodo de cabeza `(primero)`; posición `2` la correspondiente al siguiente nodo, y así sucesivamente.

El algoritmo de búsqueda comienza inicializando `indice` al nodo cabeza de la lista `(primero` de la clase `Lista)`. El bucle que se diseña en cada iteración mueve `indice` un nodo hacia adelante. El bucle termina cuando se alcanza la posición deseada e `indice` apunta al nodo correcto. El bucle también puede terminar si `indice` apunta a `null` como consecuencia de que la posición solicitada es mayor que el número de nodos de la lista.

Código Java

```
public Nodo buscarPosicion(int posicion)
{
   Nodo indice;
   int i;
   if (posicion < 0)     // posición ha de ser mayor que 0
      return null;
   indice = primero;
   for (i = 1 ;(i < posicion) && (indice != null); i++)
      indice = indice.enlace;
   return indice;
}
```

8.7. ELIMINACIÓN DE UN NODO DE UNA LISTA

La operación de eliminar un nodo de una lista enlazada supone enlazar el nodo anterior con el nodo siguiente al que se desea eliminar y liberar la memoria que ocupa. El algoritmo para eliminar un nodo que contiene un dato sigue estos pasos:

1. Búsqueda del nodo que contiene el dato. Se ha de obtener la dirección del nodo a eliminar y la dirección del anterior.
2. El `enlace` del nodo anterior que apunte al siguiente nodo del cual se elimina.
3. Si el nodo a eliminar es el *cabeza* de la lista `(primero)`, se modifica `primero` para que tenga la dirección del siguiente nodo.
4. Por último, la memoria ocupada por el nodo se libera. Es el propio sistema el que libera el nodo, al dejar de estar referenciado.

A continuación se escribe el método `eliminar()` miembro de la clase `Lista`, que recibe el dato del nodo que se quiere borrar. No utiliza el método `buscarLista` sino que desarrolla su bucle de búsqueda con el fin de disponer de la dirección del nodo anterior.

Código Java

```
public void eliminar (Elemento entrada)
{
   Nodo actual, anterior;
   boolean encontrado;
   //inicializa los apuntadores
   actual = primero;
   anterior = null;
   encontrado = false;
   // búsqueda del nodo y del anterior
   while ((actual != null) && (!encontrado))
   {
      encontrado = (actual.dato == entrada);
      //con objetos: actual.dato.equals(entrada)
      if (!encontrado)
      {
         anterior = actual;
         actual = actual.enlace;
      }
   }
   // Enlace del nodo anterior con el siguiente
   if (actual != null)
   {
      // Distingue entre que el nodo sea el cabecera,
      // o del resto de la lista
      if (actual == primero)
      {
         primero = actual.enlace;
      }
      else
      {
         anterior.enlace = actual.enlace;
      }
      actual = null; // no es necesario al ser una variable local
   }
}
```

8.8. LISTA ORDENADA

Los elementos de una lista tienen la propiedad de estar ordenados de forma lineal según las posiciones que ocupan en la misma. Se dice que n_i precede a n_{i+1} para `i = 1 ... n-1`; y que n_i sucede a n_{i-1} para `i = 2 ...` Ahora bien, también es posible mantener una lista enlazada ordenada según el dato asociado a cada nodo. La Figura 8.9 muestra una lista enlazada de números reales ordenada de forma creciente y la Figura 8.10 muestra la inserción de un elemento en dicha lista manteniendo el orden creciente de la misma. La forma de insertar un elemento en una lista ordenada es determinar, en primer lugar, la posición de inserción y, a continuación, ajustar los enlaces.

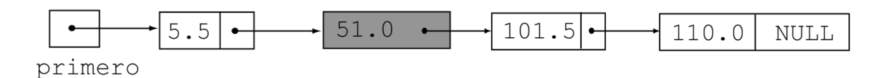


Figura 8.9 Lista ordenada

Por ejemplo, para insertar el dato `104` en la lista de la Figura 8.9 es necesario recorrer la lista hasta el nodo con `110.0`, que es el nodo inmediatamente mayor. El puntero `indice` se queda con la dirección del nodo anterior, a partir del cual se enlaza el nuevo nodo.

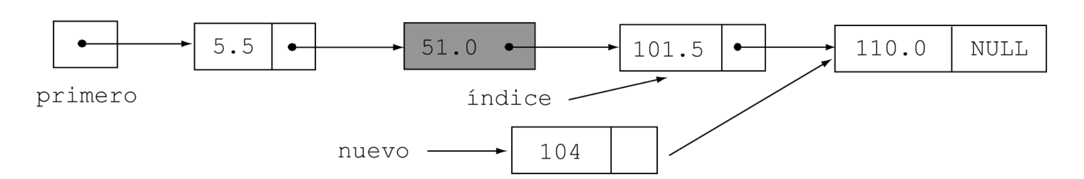


Figura 8.10 Inserción en una lista ordenada

El método `insertaOrden()` crea una lista ordenada. El punto de partida es una lista vacía, a la que se añaden nuevos elementos, de tal forma que en todo momento los elementos están ordenados en orden creciente. La inserción del primer nodo de la lista consiste, sencillamente, en crear el nodo y asignar su referencia a la cabeza de la lista. El segundo elemento se ha de insertar antes o después del primero, dependiendo de que sea menor o mayor.

En general, para insertar un nuevo elemento a la lista ordenada, se busca primero la posición de inserción en la lista actual; es decir, el nodo a partir del cual se ha de enlazar el nuevo nodo para que la lista mantenga la ordenación.

Los datos de una lista ordenada han de ser de tipo ordinal (tipo al que se pueda aplicar los operadores `==, <, >`); o bien objetos de clases que tengan definidos métodos de comparación (`equals(), compareTo(), ...`). A continuación se escribe el código Java que implementa el método para una lista de enteros y se supone que `primero` es la cabeza de la lista.

```
public ListaOrdenada insertaOrden(int entrada)
{
   Nodo nuevo ;
   nuevo = new Nodo(entrada);
   if (primero == null) // lista vacía
      primero = nuevo;
```

```
    else if (entrada < primero.getDato())
    {
      nuevo. setEnlace(primero);
      primero = nuevo;
    }
    else   /* búsqueda del nodo anterior a partir del que
              se debe insertar */
    {
      Nodo anterior, p;
      anterior = p = primero;
      while ((p.getEnlace() != null) && (entrada > p.getDato()))
      {
        anterior = p;
        p = p.getEnlace();
      }
      if (entrada > p.getDato()) //se inserta después del último nodo
        anterior = p;

      // Se procede al enlace del nuevo nodo
      nuevo.setEnlace(anterior.getEnlace());
      anterior.setEnlace(nuevo);
    }
    return this;
  }
```

8.8.1. Clase `ListaOrdenada`

Una lista enlazada ordenada *es una* lista enlazada a la que se añade la propiedad de ordenación de sus datos. Ésa es la razón que aconseja declarar la clase `ListaOrdenada` como extensión (derivada) de la clase `Lista`. Por consiguiente, hereda las propiedades de esta clase. Los métodos `eliminar()` y `buscarLista()` se redefinen para que la búsqueda del elemento aproveche el hecho de que estos están ordenados. El Ejercicio 8.2 muestra cómo se declara la clase `ListaOrdenada`.

Ejercicio 8.2

Crear una lista enlazada ordenada de números enteros. Una vez creada la lista, se recorre para escribir los datos por pantalla.

La clase `ListaOrdenada` se declara derivada de la clase `Lista`, para que herede las propiedades y métodos de `Lista`. La nueva clase contiene el método que añade elementos en orden creciente: `insertaOrden()`(escrito en el anterior apartado). El método `visualizar()`que recorre la lista y muestra los elementos, se hereda de `Lista` y no es necesario modificarlo. Los métodos que realizan búsqueda de elementos *(eliminar, buscar)* no se escriben, el lector puede redefinirlos para aprovechar el hecho de que la lista está ordenada y aumentar la eficiencia de la operación.

Con el objeto de dar entrada a los números enteros se crea una instancia de la clase `Random` para que los números se generen aleatoriamente.

```
public class ListaOrdenada extends Lista
{
  public ListaOrdenada()
```

```
{
   super();
   }
   public ListaOrdenada insertaOrden(int entrada)
   {
      /*
         este método está escrito en el apartado 8.8
      */
   }
      // métodos a codificar:
   public void eliminar (int entrada){ ; }
   public Nodo buscarLista(int destino){ ; }
}

/*
   clase con el método main
*/
import java.util.*;
import ListaEnteros.ListaOrdenada;

public class ListaEnOrden
{
   public static void main(String [] a)
   {
      Random r;
      int d;
      ListaOrdenada lista;
      int k;
      r = new Random();             // generador de números aleatorios
      lista = new ListaOrdenada();  // crea lista vacía
      k = r.nextInt(99)+1;          // número de elementos
      // inserta elementos en la lista
      for (; k >= 0; k-- )
      {
         d = r.nextInt();
        lista.insertaOrden(d);
      }
      // escribe los elementos de la lista
      System.out.println("Elementos de la lista ordenada \n");
      lista.visualizar();
   }
}
```

8.9. LISTA DOBLEMENTE ENLAZADA

Hasta ahora, el recorrido de una lista se ha realizado en sentido directo *(adelante)*. Existen numerosas aplicaciones en las que es conveniente poder acceder a los elementos o nodos de una lista en cualquier orden, tanto hacia adelante como hacia atrás. En este caso, se recomienda el uso de una **lista doblemente enlazada.** En esta lista, cada elemento contiene dos punteros (referencias), además del valor almacenado. Una referencia apunta al siguiente elemento de la lista y la otra

referencia apunta al elemento anterior. La Figura 8.11 muestra una lista doblemente enlazada y un nodo de dicha lista.

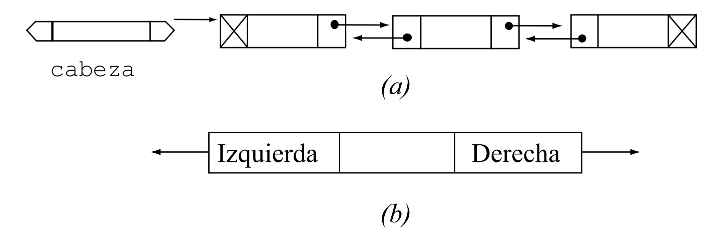


Figura 8.11 Lista doblemente enlazada. (a) Lista con tres nodos; (b) nodo

Las operaciones de una *Lista Doble* son similares a las de una *Lista: insertar, eliminar, buscar, recorrer...* La operación de insertar un nuevo nodo en la lista debe realizar ajustes de los dos *pointer*. La Figura 8.12 muestra el problema de insertar un nodo a la derecha del nodo *actual;* como se observa, se asignan cuatro enlaces.

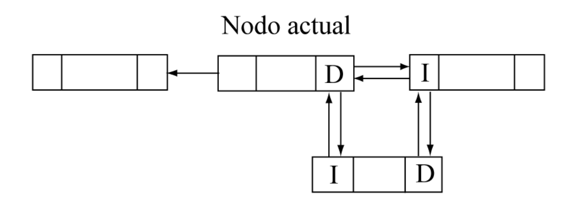


Figura 8.12 Inserción de un nodo en una lista doblemente enlazada

La operación de eliminar (borrar) un nodo de la lista doble necesita enlazar, mutuamente, el nodo anterior y el nodo siguiente del que se borra, como se observa en la Figura 8.13.

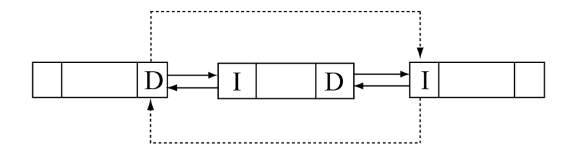


Figura 8.13 Eliminación de un nodo en una lista doblemente enlazada

8.9.1. Nodo de una lista doblemente enlazada

Un nodo de una lista doblemente enlazada tiene dos punteros (referencias) para enlazar con los nodos izquierdo y derecho, además de la parte correspondiente al campo dato. La clase `Nodo` agrupa los componentes del nodo de una lista doble; por ejemplo, para datos de tipo entero:

```
package listaDobleEnlace;

public class Nodo
{
   int dato;
```

```
   Nodo adelante;
   Nodo atras;
      // ...
}
```

El constructor asigna un valor al campo `dato` y las referencias `adelante`, `atras` se inicializan a `null`.

```
public Nodo(int entrada)
{
   dato = entrada;
   adelante = atras = null;
}
```

8.9.2. Insertar un elemento en una lista doblemente enlazada

La clase `ListaDoble` encapsula las operaciones básicas de las listas doblemente enlazadas. La clase dispone de la variable `cabeza` que referencia el primer nodo de la lista, permite acceder a cualquier otro nodo. El constructor de la clase inicializa la lista vacía (`null`).

Se puede añadir un nuevo nodo a la lista de distintas formas, según la posición donde se inserte. Naturalmente, el algoritmo empleado para añadir varía dependiendo de la posición en que se desea insertar el elemento. La posición de inserción puede ser:

- En la *cabeza* (elemento primero) de la lista.
- Al final de la lista (elemento último).
- Antes de un elemento especificado.
- Después de un elemento especificado.

Insertar un nuevo elemento en la cabeza de una lista doble

El proceso sigue estos pasos:

1. Crear un nodo con el nuevo elemento y asignar su referencia a la variable `nuevo`.
2. Hacer que el campo enlace `adelante` del nuevo nodo apunte a la cabeza (primer nodo) de la lista original, y que el campo enlace `atras` del nodo cabeza apunte al nuevo nodo.
3. Hacer que `cabeza` apunte al nuevo nodo que se ha creado.

A continuación, se escribe el método, miembro de la clase `ListaDoble`, que implementa la operación.

Código Java

```
public ListaDoble insertarCabezaLista(Elemento entrada)
{
   Nodo nuevo;
   nuevo = new Nodo(entrada);
   nuevo.adelante = cabeza;
   if (cabeza != null )
     cabeza.atras = nuevo;
   cabeza = nuevo;
   return this;
}
```

Insertar después de un nodo

La inserción de un nuevo nodo se puede realizar también en un nodo intermedio de la lista. El algoritmo de la operación que inserta después de un nodo `n`, requiere las siguientes etapas:

1. Crear un nodo con el nuevo elemento y asignar su referencia a la variable `nuevo`.
2. Hacer que el enlace `adelante` del nuevo nodo apunte al nodo siguiente de `n` (o bien a `null` si `n` es el último nodo). El enlace `atras` del nodo siguiente a `n` (si `n` no es el último nodo) tiene que apuntar a `nuevo`.
3. Hacer que el enlace `adelante` del nodo `n` apunte al nuevo nodo. A su vez, el enlace `atras` del nuevo nodo debe de apuntar a `n`.

El método `insertaDespues()` implementa el algoritmo, supone que la lista es de números enteros, y naturalmente es un método de la clase `ListaDoble`. El primer argumento, `anterior`, representa el nodo `n` a partir del cual se enlaza. El segundo argumento, `entrada`, es el dato que se añade a la lista.

Código Java

```
public ListaDoble insertaDespues(Nodo anterior, Elemento entrada)
{
   Nodo nuevo;

   nuevo = new Nodo(entrada);
   nuevo.adelante = anterior.adelante;
   if (anterior.adelante != null)
      anterior.adelante.atras = nuevo;
   anterior.adelante = nuevo;
   nuevo.atras = anterior;
   return this;
}
```

8.9.3. Eliminar un elemento de una lista doblemente enlazada

Quitar un nodo de una lista doble supone realizar el enlace de dos nodos, el nodo *anterior* y el nodo *siguiente* al que se desea eliminar. La referencia `adelante` del nodo anterior debe apuntar al nodo *siguiente,* y la referencia `atras` del nodo *siguiente* debe apuntar al nodo *anterior.* La memoria que ocupa el nodo se libera automáticamente en el momento que éste deja de ser referenciado *(garbage collection*, recolección de basura).

El algoritmo es similar al del borrado para una lista simple. Ahora, la dirección del nodo *anterior* se encuentra en la referencia `atras` del nodo a borrar. Los pasos a seguir son:

1. Búsqueda del nodo que contiene el dato.
2. La referencia `adelante` del nodo anterior tiene que apuntar a la referencia `adelante` del nodo a eliminar (si no es el nodo *cabecera*).
3. La referencia `atras` del nodo siguiente a borrar tiene que apuntar a la referencia `atras` del nodo a eliminar (si no es el último nodo).
4. Si el nodo que se elimina es el primero, `cabeza`, se modifica `cabeza` para que tenga la dirección del nodo `siguiente`.
5. La memoria ocupada por el nodo es liberada automáticamente.

El método que implementa el algoritmo es miembro de la clase `ListaDoble`:

```
public void eliminar (Elemento entrada)
{
   Nodo actual;
   boolean encontrado = false;
   actual = cabeza;
   // Bucle de búsqueda
   while ((actual != null) && (!encontrado))
   {
        /* la comparación se realiza con el método equals()...,
         depende del tipo Elemento */
      encontrado = (actual.dato == entrada);
      if (!encontrado)
         actual = actual.adelante;
   }
    // Enlace de nodo anterior con el siguiente
   if (actual != null)
   {
      //distingue entre nodo cabecera o resto de la lista
      if (actual == cabeza)
      {
         cabeza = actual.adelante;
         if (actual.adelante != null)
            actual.adelante.atras = null;
      }
      else if (actual.adelante != null)   // No es el último nodo
      {
         actual.atras.adelante = actual.adelante;
         actual.adelante.atras = actual.atras;
      }
      else  // último nodo
         actual.atras.adelante = null;
      actual = null;
   }
}
```

Ejercicio 8.3

Crear una lista doblemente enlazada con números enteros, del 1 al 999, generados aleatoriamente. Una vez creada la lista, se eliminan los nodos que estén fuera de un rango de valores leídos desde el teclado.

En el apartado 8.9 se han declarado las clases `Nodo` y `ListaDoble`, formando parte del paquete `listaDobleEnlace`, necesarias para este ejercicio. Además, a la clase `Nodo` se añade el método `getDato()` para devolver el dato del nodo.

Para resolver el problema que plantea el ejercicio, en el paquete `listaDobleEnlace` se define la clase `IteradorLista`, para acceder a los datos de cualquier lista doble (de enteros). En el constructor de la clase `IteradorLista` se asocia la lista a recorrer con el objeto iterador. La clase `IteradorLista` implementa el método `siguiente()`; cada llamada a `siguiente()` devuelve el nodo actual de la lista y avanza al siguiente. Una vez que se termina de recorrer la lista, devuelve `null`.

La clase con el método `main()` crea un objeto que genera números aleatorios, los cuales se insertan en la lista doble. A continuación, se pide el rango de elementos a eliminar; con el objeto iterador se obtienen los elementos, y aquellos fuera de rango se borran de la lista (llamada al método `eliminar`).

```
// archivo con la clase Nodo de visibilidad public
package listaDobleEnlace;

public class Nodo
{
   // declaración de nodo de lista doble
   public int getDato()
   {
      return dato;
   }
}

// archivo con la clase ListaDoble de visibilidad public

package listaDobleEnlace;

public class ListaDoble
{
   Nodo cabeza;
   // métodos de la clase (implementación en apartado 8.9)
   public ListaDoble(){;}
   public ListaDoble insertarCabezaLista(int entrada){;}
   public ListaDoble insertaDespues(Nodo anterior, int entrada){;}
   public void eliminar (int entrada){;}
   public void visualizar() {;}
   public void buscarLista(int destino) {;}

}

// archivo con la clase IteradorLista de visibilidad public

package listaDobleEnlace;

public class IteradorLista
{
   private Nodo actual;
   public IteradorLista(ListaDoble ld)
   {
      actual = ld.cabeza;
   }
   public Nodo siguiente()
   {
      Nodo a;
      a = actual;
      if (actual != null)
      {
         actual = actual.adelante;
      }
      return a;
   }
}
```

```
/*
   Clase con método main(). Crea el objeto lista doble e inserta
   datos enteros generados aleatoriamente.
   Crea objeto iterador de lista, para recorrer sus elementos y
   aquellos fuera de rango se eliminan. El rango se lee del teclado.
*/

import java.util.Random;
import java.io.*;
import listaDobleEnlace.*;

class ListaEnRango
{
   public static void main(String [] ar) throws IOException
   {
      Random r;
      int d, x1, x2;
      final int M = 29;  // número de elementos de la lista
      final int MX = 999;
      BufferedReader entrada = new BufferedReader(
                               new InputStreamReader(System.in));
      ListaDoble listaDb;
      r = new Random();
      listaDb = new ListaDoble();
      for (int j = 1; j <= M ; j++)
      {
         d = r.nextInt(MX) + 1;
         listaDb.insertarCabezaLista(d);
      }

      System.out.println("Elementos de la lista original");
      listaDb.visualizar();
      // rango de valores
      System.out.println("\nRango que va a contener la lista");
      x1 = Integer.parseInt(entrada.readLine());
      x2 = Integer.parseInt(entrada.readLine());
      // crea iterador asociado a la lista
      IteradorLista iterador = new IteradorLista(listaDb);
      Nodo a;
      // recorre la lista con el iterador
      a = iterador.siguiente();
      while (a != null)
      {
         int w;
         w = a.getDato();
         if (!(w >= x1 && w <= x2))  // fuera de rango
            listaDb.eliminar(w);
         a = iterador.siguiente();
      }

      System.out.println("Elementos actuales de la lista");
      listaDb.visualizar();
   }
}
```

8.10. LISTAS CIRCULARES

En las listas lineales simples o en las dobles siempre hay un primer nodo (cabeza) y un último nodo (cola). Una lista circular, por propia naturaleza, no tiene ni principio ni fin. Sin embargo, resulta útil establecer un nodo a partir del cual se acceda a la lista y así poder acceder a sus nodos.

La Figura 8.14 muestra una lista circular con enlace simple; podría considerarse que es una lista lineal cuyo último nodo apunta al primero.

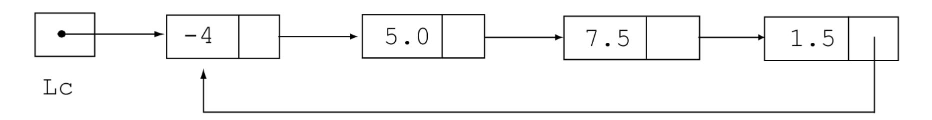


Figura 8.14 Lista circular

Las operaciones que se realizan sobre una lista circular son similares a las operaciones sobre listas lineales, teniendo en cuenta que no hay *primero* ni *último* nodo, aunque sí un nodo de acceso a la lista. Estas operaciones permiten construir el *TAD* `ListaCircular` y su funcionalidad es la siguiente:

- Inicialización o creación.
- Inserción de elementos en una lista circular.
- Eliminación de elementos de una lista circular.
- Búsqueda de elementos de una lista circular.
- Recorrido de cada uno de los nodos de una lista circular.
- Verificación de lista vacía.

La construcción de una lista circular se puede hacer con enlace simple o enlace doble. La implementación que se desarrolla en este apartado enlaza dos nodos con un enlace simple.

Se declara la clase `Nodo`, con el campo dato y enlace, y la clase `ListaCircular` con el puntero de acceso a la lista, junto a los métodos que implementan las operaciones. Los elementos de la lista pueden ser de cualquier tipo, se puede abstraer su tipo en otra clase, por ejemplo `Elemento`; con el fin de simplificar, se supone un tipo conocido.

El constructor de la clase `Nodo` varía respecto al de las listas no circulares, ya que el campo referencia `enlace`, en vez de quedar a `null`, se inicializa para que apunte al mismo nodo, de tal forma que queda como lista circular de un solo nodo.

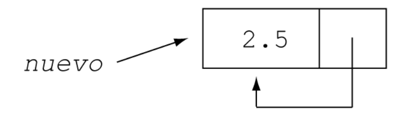


Figura 8.15 Creacción de un nodo de lista circular.

```
package listaCircular;

public class Nodo
{
   Elemento dato;
   Nodo enlace;
```

```
    public Nodo (Elemento entrada)
    {
       dato = entrada;
       enlace = this; // se apunta asímismo
    }
 }
```

A tener en cuenta

El pointer de acceso a una lista circular, lc, normalmente apunta al último nodo añadido a la estructura. Esta convención puede cambiar, ya que en una estructura circular no hay *primero* ni *último*.

8.10.1. Insertar un elemento en una lista circular

El algoritmo empleado para añadir o insertar un elemento en una lista circular varía dependiendo de la posición en que se desea insertar. La posición de inserción puede variar. Consideramos que se hace como nodo anterior al del nodo de acceso a la lista `lc`, y que `lc` tiene la dirección del último nodo insertado. A continuación, se declara la clase `ListaCircular` con el constructor *(lista vacía)* y el método de inserción.

```
 package listaCircular;

 public class ListaCircular
 {
    private Nodo lc;

    public ListaCircular()
    {
       lc = null;
    }
    public ListaCircular insertar(Elemento entrada)
    {
      Nodo nuevo;
      nuevo = new Nodo(entrada);
      if (lc != null) // lista circular no vacía
      {
         nuevo.enlace = lc.enlace;
         lc.enlace = nuevo;
      }
       lc = nuevo;
       return this;
    }
    //...
```

8.10.2. Eliminar un elemento en una lista circular

La operación de eliminar un nodo de una lista circular sigue los mismos pasos que los dados para eliminar un nodo en una lista lineal. Hay que enlazar el nodo anterior con el nodo siguiente al que se desea eliminar y que el sistema libere la memoria que ocupa.

El algoritmo para eliminar un nodo de una lista circular es:

1. Búsqueda del nodo que contiene el dato.
2. Se enlaza el nodo anterior con el siguiente.
3. En caso de que el nodo a eliminar sea por el que se accede a la lista, `lc`, se modifica `lc` para que tenga la dirección del nodo anterior.
4. Por último, el sistema libera la memoria ocupada por el nodo al anular la referencia.

La implementación del método debe de tener en cuenta que la lista circular conste de un sólo nodo, ya que al eliminarlo la lista se queda vacía `(lc = null)`. La condición de lista con un nodo se corresponde con la forma de inicializar (constructor) un nodo: `lc == lc.enlace`.

El método recorre la lista buscando el nodo con el dato a eliminar, utiliza un puntero al nodo *anterior* para que cuando se encuentre el nodo se enlace con el *siguiente*. Se accede al dato con la sentencia `actual.enlace.dato`; éste permite, si coincide con el dato a eliminar, tener en `actual` el nodo anterior. Después del bucle es necesario volver a preguntar por el campo `dato`, ya que no se comparó el nodo de acceso a la lista, `lc`, y el bucle puede terminar sin encontrar el nodo.

Código Java

```
public void eliminar(Elemento entrada)
{
   Nodo actual;
   boolean encontrado = false;
   //bucle de búsqueda
   actual = lc;
   while ((actual.enlace != lc) && (!encontrado))
   {
      encontrado = (actual.enlace.dato == entrada);
      if (!encontrado)
      {
         actual = actual.enlace;
      }
   }
   encontrado = (actual.enlace.dato == entrada);

   // Enlace de nodo anterior con el siguiente

   if (encontrado)
   {
      Nodo p;
      p = actual.enlace;    // Nodo a eliminar
      if (lc == lc.enlace)  // Lista con un solo nodo
         lc = null;
      else
      {
         if (p == lc)
         {
            lc = actual;  // Se borra el elemento referenciado por lc,
                          // el nuevo acceso a la lista es el anterior
         }
         actual.enlace = p.enlace;
      }
      p = null;
   }
}
```

8.10.3. Recorrer una lista circular

Una operación común a todas las estructuras enlazadas es recorrer o visitar todos los nodos de la estructura. En una lista circular, el recorrido puede empezar en cualquier nodo e ir procesando iterativamente cada nodo hasta alcanzar el de partida. El método (miembro de la clase `ListaCircular`) que se va escribir inicia el recorrido en el nodo siguiente al de acceso a la lista, `lc`, termina cuando alcanza el nodo `lc`. El proceso que se realiza con cada nodo consiste en escribir su contenido.

```
public void recorrer()
{
   Nodo p;
   if (lc != null)
   {
      p = lc.enlace; // siguiente nodo al de acceso
      do {
         System.out.println("\t" + p.dato);
         p = p.enlace;
      }while (p != lc.enlace);
   }
   else
      System.out.println("\t Lista Circular vacía.");
}
```

Ejercicio 8.4

Crear una lista circular con palabras leídas del teclado. El programa debe tener un conjunto de opciones:

- *Mostrar las cadenas que forman la lista.*
- *Borrar una palabra dada.*
- *Al terminar la ejecución, recorrer la lista eliminando los nodos.*

Para crear la lista circular, se utilizan las clases `Nodo` y `ListaCircular` del paquete `listaCircularPalabra` y la clase con el método `main()`. Al ser una lista de palabras, el campo dato del nodo es de tipo `String`.

El método `readLine()` es el apropiado para leer una cadena desde el teclado; cada llamada a `readLine()` crea un objeto cadena con la palabra leída, que a su vez será el nuevo dato de la lista circular. La inserción en la lista se hace llamando al método de `ListaCircular insertar()`.

Para borrar una palabra, se llama al método `eliminar()`; este busca el nodo que tiene la palabra, utiliza el método `equals()`, en lugar del operador ==, para determinar si coincide la palabra buscada con la del nodo.

Con el fin de recorrer la lista circular liberando cada nodo, se declara el método `borrarLista()` en la clase `ListaCircular`.

Consulte el apartado 8.10 para conocer el detalle de los método de las clases `Nodo` y `ListaCircular`; ahora sólo se escriben las pequeñas diferencias y el método `borrarLista()`.

```
package listaCircularPalabra;

class Nodo
{
```

```
   String dato;
   Nodo enlace;
   public Nodo (String entrada) {; }
}

public class ListaCircular
{
   private Nodo lc;

   public ListaCircular(){;}
   public ListaCircular insertar(String entrada){;}
   public void eliminar(String entrada)
   {
      Nodo actual;

      actual = lc;
      while ((actual.enlace != lc) &&
                        !(actual.enlace.dato.equals(entrada)))
      {
         if (!actual.enlace.dato.equals(entrada))
            actual = actual.enlace;
      }
         // Enlace de nodo anterior con el siguiente
         // si se ha encontrado el nodo.

      if (actual.enlace.dato.equals(entrada))
      {
         Nodo p;
         p = actual.enlace;         // Nodo a eliminar
         if (lc == lc.enlace)       // Lista con un solo nodo
            lc = null;
         else
         {
            if (p == lc)
            {
               lc = actual; // Se borra el elemento referenciado por lc,
                            // el nuevo acceso a la lista es el anterior
            }
            actual.enlace = p.enlace;
         }
         p = null;
      }
   }

   public void borrarLista()
   {
      Nodo p;
      if (lc != null)
      {
         p = lc;
         do {
            Nodo t;
            t = p;
            p = p.enlace;
            t = null;  // no es estrictamente necesario
         }while(p != lc);
      }
      else
```

```
         System.out.println("\n\t Lista vacía.");
      lc = null;
   }
   public void recorrer(){;}
}
/* clase con el método main().  Se escribe un sencillo menu para
   elegir operaciones con la lista circular.
*/
import java.io.*;
import listaCircularPalabra.*;
class ListaPalabras
{
   public static void main(String [] a) throws IOException
   {
      String palabra;
      ListaCircular listaCp;
      int opc;
      BufferedReader entrada = new BufferedReader(
                              new InputStreamReader(System.in));

      listaCp = new ListaCircular();
      System.out.println("\n Entrada de Nombres. Termina con ^Z.\n");
      while ((palabra = entrada.readLine())!= null)
      {
         String nueva;
         nueva = new String(palabra);
         listaCp.insertar(nueva);
      }
      System.out.println("\t\tLista circular de palabras");
      listaCp.recorrer();

      System.out.println("\n\t Opciones para manejar la lista");
      do {
         System.out.println("1. Eliminar una palabra.\n");
         System.out.println("2. Mostrar la lista completa.\n");
         System.out.println("3. Salir y eliminar la lista.\n");
         do {
            opc = Integer.parseInt(entrada.readLine());
        }while (opc<1 || opc>3);

         switch (opc) {
         case 1: System.out.print("Palabra a eliminar: ");
                 palabra = entrada.readLine();
                 listaCp.eliminar(palabra);
                 break;
         case 2: System.out.println("Palabras en la Lista:\n");
                 listaCp.recorrer();
                 break;
         case 3: System.out.print("Eliminación de la lista.");
                 listaCp.borrarLista();
         }
      }while (opc != 3);
   }
}
```

8.11. LISTAS ENLAZADAS GENÉRICAS

La definición de una lista está muy ligada al tipo de datos de sus elementos; así, se han puesto ejemplos en los que el tipo es `int`, otros en los que el tipo es `double`, otros `String`. Declarando el campo dato de tipo `Object` se consigue una lista genérica, válida para cualquier tipo de dato, aunque exigirá muchas conversiones de datos cuando se concrete para un tipo de dato particular.

Es preciso recordar que la clase `Object` (incuida en el paquete `java.lang`) es la clase base de cualquier clase no derivada. Cualquier referencia a un objeto se puede asignar a una variable de tipo `Object`. Entonces, si se define el campo dato de tipo `Object`, la conversión automática permite crear listas enlazadas de cualquier tipo de objetos, aunque, siempre tendrán que ser listas de objetos. Para manejar una lista cuyos datos son de un *tipo simple*, por ejemplo, entero (`int`), es preciso crear objetos de la clase *envolvente* que le correspondan; en el caso del tipo `int` la clase `Integer`.

Ejemplo 8.5

Se definen referencias a objetos de String, Integer y se asignan a una referencia a Object.

```
String gr = "Objeto cadena";
Integer t = new Integer(5);
Object q;

q = t;      // asignación correcta
q = gr;     // asignación correcta
gr = q;     // error, no se puede asignar una subclase a clase base
```

8.11.1. Declaración de la clase lista genérica

Las operaciones del tipo lista genérica son las mismas que las definidas en el apartado 8.4. En algunos métodos varía la implementación, debido a que la comparación de dos elementos no se hace con el operador ==, sino con el método `equals()`.

```
// Declaración de la clase nodo

package listaGenerica;

public class Nodo
{
   Object dato;
   Nodo enlace;

   public Nodo(Object x)
   {
      dato = x;
      enlace = null;
   }
      public Nodo(Object x, Nodo n)
   {
      dato = x;
      enlace = n;
```

```
    }
    public Object leerDato()
    {
        return dato;
    }
    public Nodo siguiente()
    {
        return enlace;
    }
}
    // Declaración de la clase Lista.
package listaGenerica;

public class Lista
{
    private Nodo primero;

    public Lista()
    {
        primero = null;
    }

    public Nodo leerPrimero()
    {
        return primero;
    }
    public Lista insertarCabezaLista(Object entrada)
    {
        Nodo nuevo;
        nuevo = new Nodo(entrada)
        nuevo.enlace = primero;
        primero= nuevo;
        return this;
    }
    // inserta un elemento a partir de anterior
    public Lista insertarLista(Nodo anterior, Object entrada)
    {
        Nodo nuevo;
        nuevo = new Nodo(entrada);
        nuevo.enlace = anterior.enlace
        anterior.enlace = nuevo;
        return this;
    }
    // elimina el elemento entrada
    public void eliminar (Object entrada)
    {
        Nodo actual, anterior;
        boolean encontrado;
        actual = primero;
        anterior = null;
        encontrado = false;
        // Bucle de búsqueda
        while ((actual!= null) && !actual.dato.equals(entrada))
```

```
        {
           if (!actual.dato.equals(entrada))
           {
              anterior = actual;
              actual = actual.enlace;
           }
        }
        if (actual != null)
        {
           // Se distingue entre que el nodo sea el cabecera
           // o del resto de la lista
           if (acual == primero)
           {
              primero = actual.enlace;
           }
           else
           {
              anterior.enlace = actual.enlace
           }
           actual = null;
        }
     }
     // busca el elemento destino
     public Nodo buscarLista(Object destino)
     {
        Nodo indice;

        for (indice = primero; indice != null; indice = indice.enlace)
           if (indice.dato.equals(destino))
              return indice;
        return null;
     }
     // recorre la lista y muestra cada dato
     public void visualizar()
     {
        Nodo n;
        n = primero;
        while (n != null)
        {
           Sistem.out.print(n.dato + "  ");
           n = n.enlace;
        }
     }
  }
```

8.11.2. Iterador de lista

Un objeto *Iterador* se diseña para recorrer los elementos de un contenedor. Un iterador de una lista enlazada accede a cada uno de sus nodos, hasta alcanzar el último elemento. El constructor del objeto iterador inicializa el puntero *actual* al primer elemento de la estructura; el método `siguiente()` devuelve el elemento *actual* y hace que éste quede apuntando al siguiente elemento; si no existe, devuelve `null`.

La clase `ListaIterador` implementa el iterador de una lista enlazada genérica, su constructor inicializa `actual` al primer nodo. El método `siguiente()` devuelve la referencia al *objeto dato* del nodo y avanza al siguiente nodo.

```
package listaGenerica;

public class ListaIterador
{
   private Nodo prm, actual;

   public ListaIterador(Lista list)
   {
      prm = actual = list.leerPrimero();
   }

   public Object siguiente()
   {
      Object elemento = null;
      if (actual != null)
      {
         elemento = actual.leerDato();
         actual = actual.siguiente();
      }
      return elemento;
   }
   public void inicIter()
   {
      actual = prm;
   }
}
```

RESUMEN

Una **lista enlazada** es una estructura de datos dinámica en la cual sus componentes están ordenados lógicamente por sus campos de enlace, en vez de ordenados físicamente como en un *array*. El final de la lista se señala mediante una constante o referencia especial llamada `null`. La gran ventaja de una lista enlazada sobre un *array* es que puede crecer y decrecer en tamaño, ajustándose al número de elementos.

Una **lista simplemente enlazada** contiene sólo un enlace a un sucesor único, a menos que sea el último, en cuyo caso no se enlaza con ningún otro nodo.

Cuando se inserta un elemento en una lista enlazada, se deben considerar cuatro casos: añadir a una lista vacía, añadir al principio de la lista, añadir en el interior y añadir al final de la lista.

Para borrar un elemento, primero hay que buscar el nodo que lo contiene y considerar dos casos: borrar el primer nodo y borrar cualquier otro nodo de la lista.

El recorrido de una lista enlazada significa pasar por cada nodo (visitar) y procesarlo. El proceso de cada nodo puede consistir en escribir su contenido, modificar el campo dato...

Una **lista doblemente enlazada** es aquella en la que cada nodo tiene una referencia a su sucesor y otra a su predecesor. Las listas doblemente enlazadas se pueden recorrer en ambos sentidos. Las operaciones básicas son inserción, borrado y recorrer la lista, similares a las de las listas simples.

Una **lista enlazada circularmente** por propia naturaleza no tiene primero ni último nodo. Las listas circulares pueden ser de enlace simple o doble.

Una **lista enlazada genérica** tiene como tipo de dato `Object`. Al ser `Object` la *super clase base* de cualquier clase no derivada, con la clase `ListaGenerica` se pueden crear listas de cualquier tipo de dato referencia.

EJERCICIOS

8.1. Escribir un método, en la clase `Lista`, que devuelva cierto si la lista está vacía.

8.2. Añadir a la clase `ListaDoble` un método que devuelva el número de nodos de una lista doble.

8.3. En una lista enlazada de números enteros, se desea añadir un nodo entre dos nodos consecutivos cuyos datos tienen distinto signo; el dato del nuevo nodo debe ser la diferencia, en valor absoluto, de los dos nodos.

8.4. Añadir a la clase `Lista` el método `eliminaraPosicion()` para que elimine el nodo que ocupa la posición `i`, siendo el nodo cabecera el que ocupa la posición `0`.

8.5. Escribir un método que tenga como argumento la referencia `cabeza` al primer nodo de una lista enlazada. El método crea una lista doblemente enlazada con los mismos campos dato pero en orden inverso.

8.6. Se tiene una lista simplemente enlazada de números reales. Escribir un método para obtener una lista doble ordenada respecto al campo dato con los valores reales de la lista simple.

8.7. Escribir un método para crear una lista doblemente enlazada de palabras introducidas por teclado. La referencia de acceso a la lista debe ser el nodo que está en la posición intermedia.

8.8. La clase `ListaCircularCadena` dispone de los métodos que implementan las operaciones de una lista circular de palabras. Escribir un método que cuente el número de veces que una palabra dada se encuentra en la lista.

8.9. Dada una lista circular de números enteros, escribir un método que devuelva el mayor entero de la lista.

8.10. Se tiene una lista enlazada donde el campo dato es objeto alumno con las variables: *nombre, edad, sexo*. Escribir un método para transformar la lista de tal forma que si el primer nodo es un alumno de sexo masculino, el siguiente sea de sexo femenino, y así alternativamente, siempre que sea posible.

8.11. Una lista circular de cadenas está ordenada alfabéticamente. El puntero de acceso a la lista tiene la dirección del nodo alfabéticamente mayor. Escribir un método para añadir una nueva palabra, en el orden que le corresponda, a la lista.

8.12. Dada la lista del Ejercicio 8.11, escribir un método que elimine una palabra dada.

PROBLEMAS

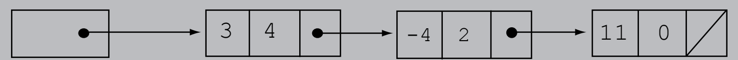
8.1 Escribir un programa que realice las siguientes tareas:

- Crear una lista enlazada de números enteros positivos al azar, donde la inserción se realiza por el último nodo.
- Recorrer la lista para mostrar los elementos por pantalla.
- Eliminar todos los nodos que superen un valor dado.

8.2. Se tiene un archivo de texto de palabras separadas por un blanco o el carácter de fin de línea. Escribir un programa para formar una lista enlazada con las palabras del archivo. Una vez formada la lista, se pueden añadir nuevas palabras o borrar alguna de ellas. Al finalizar el programa, escribir las palabras de la lista en el archivo.

8.3. Un polinomio se puede representar como una lista enlazada. El primer nodo de la lista representa el primer término del polinomio, el segundo nodo al segundo término del polinomio y así sucesivamente. Cada nodo tiene como campo dato el coeficiente del término y el exponente.

Por ejemplo, el polinomio $3x^4 – 4x^2 + 11$ se representa así:



Escribir un programa que permita dar entrada a polinomios en *x,* representándolos con una lista enlazada simple. A continuación, obtener una tabla de valores del polinomio para valores de x = 0.0, 0.5, 1.0, 1.5, … , 5.0.

8.4 Teniendo en cuenta la representación de un polinomio propuesta en el Problema 8.3, hacer los cambios necesarios para que la lista enlazada sea circular. La referencia de acceso debe tener la dirección del último término del polinomio, el cual apuntará al primer término.

8.5 Según la representación de un polinomio propuesta en el Problema 8.4, escribir un programa que realice las siguientes operaciones:

- Obtener la lista circular suma de dos polinomios.
- Obtener el polinomio derivada.
- Obtener una lista circular que sea el producto de dos polinomios.

8.6. Escribir un programa para obtener una lista doblemente enlazada con los caracteres de una cadena leída desde el teclado. Cada nodo de la lista tendrá un carácter. Una vez que se haya creado la lista, ordenarla alfabéticamente y escribirla en pantalla.

8.7. Un conjunto es una secuencia de elementos, todos ellos del mismo tipo sin duplicidades. Escribir un programa para representar un conjunto de enteros mediante una lista enlazada. El programa debe contemplar las siguientes operaciones:

- Cardinal del conjunto.
- Pertenencia de un elemento al conjunto.
- Añadir un elemento al conjunto.
- Escribir en pantalla los elementos del conjunto.

8.8. Con la representación propuesta en el Problema 8.7, añadir las operaciones básicas de conjuntos:

- Unión de dos conjuntos.
- Intersección de dos conjuntos.
- Diferencia de dos conjuntos.
- Inclusión de un conjunto en otro.

8.9. Escribir un programa en el que dados, dos archivos `F1`, `F2` formados por palabras separadas por un blanco o fin de línea, se creen dos conjuntos con las palabras de `F1` y `F2` respectivamente. Posteriormente, encontrar las palabras comunes y mostrarlas en pantalla.

8.10. Utilizar una lista doblemente enlazada para controlar una lista de pasajeros de una línea aérea. El programa principal debe ser controlado por menú y permitir al usuario visualizar los datos de un pasajero determinado, insertar un nodo (siempre por el final) y eliminar un pasajero de la lista. A la lista se accede por dos variables, una referencia al primer nodo y la otra al último nodo.

8.11. Para representar un entero largo, de más de 30 dígitos, utilizar una lista circular cuyos nodos tengan como campo dato un dígito del entero largo. Escribir un programa en el que se introduzcan dos enteros largos y se obtenga su suma.

8.12. Un vector disperso es aquél que tiene muchos elementos que son cero. Escribir un programa que permita representar mediante listas enlazadas un vector disperso. Los nodos de la lista son los elementos de la lista distintos de cero; en cada nodo se representa el valor del elemento y el índice (posición del vector). El programa ha de realizar las siguientes operaciones: sumar dos vectores de igual dimensión y hallar el producto escalar.



CAPITULO 9
Pilas

Objetivos

Con el estudio de este capítulo, usted podrá:

- Especificar el tipo abstracto de datos Pila.
- Conocer aplicaciones de las Pilas en la programación.
- Definir e implementar la clase Pila.
- Conocer las diferentes formas de escribir una expresión.
- Evaluar una expresión algebraica.

Contenido:

Conceptos clave

- ♦ Expresiones y sus tipos.
- ♦ Listas enlazadas.
- ♦ Notación de una expresión.
- ♦ Pila.
- ♦ Prioridad.
- ♦ Tipo abstracto de datos.

INTRODUCCIÓN

En este capítulo se estudia en detalle la estructura de datos pila, utilizada frecuentemente en los programas más usuales. Es una estructura de datos que almacena y recupera sus elementos atendiendo a un orden estricto. Las pilas se conocen también como estructuras **LIFO** *(Last-in, first-out,* último en entrar primero en salir). El desarrollo de las pilas como tipos abstractos de datos es el motivo central de este capítulo.

Las pilas se utilizan en compiladores, sistemas operativos y programas de aplicaciones. Una aplicación interesante es la evaluación de expresiones aritméticas, también la organización de la memoria.

9.1. CONCEPTO DE PILA

Una **pila** *(stack)* es una colección ordenada de elementos a los cuales sólo se puede acceder por un único lugar o extremo de la pila. Los elementos se añaden o se quitan (borran) de la pila sólo por su parte superior **(cima)**. Este es el caso de una pila de platos, una pila de libros, etc.

> ***A recordar***
>
> Una pila es una estructura de datos de entradas ordenadas que sólo se pueden introducir y eliminar por un extremo, llamado **cima.**

Cuando se dice que la pila está ordenada, lo que se quiere decir es que hay un elemento al que se puede acceder primero (el que está encima de la pila), otro elemento al que se puede acceder en segundo lugar (justo el elemento que está debajo de la cima), un tercero, etc. No se requiere que las entradas se puedan comparar utilizando el operador "menor que" (<) y pueden ser de cualquier tipo.

Las entradas de la pila deben ser eliminadas en el orden inverso al que se situaron en la misma. Por ejemplo, se puede crear una pila de libros, situando primero un diccionario, encima de él una enciclopedia y encima de ambos una novela, de modo que la pila tendrá la novela en la parte superior.

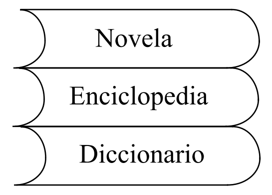


Figura 9.1 Pila de libros

Cuando se quitan los libros de la pila, primero debe quitarse la novela, luego la enciclopedia y por último el diccionario.

Debido a su propiedad específica *último en entrar, primero en salir* se conoce a las pilas como estructuras de datos **LIFO** (*last-in, first-out*)

Las operaciones usuales en la pila son *Insertar* y *Quitar*. La operación **Insertar** `(push)` añade un elemento en la cima de la pila, y la operación **Quitar** `(pop)` elimina o saca un elemento de

la pila. La Figura 9.2 muestra una secuencia de operaciones *Insertar* y *Quitar*. El último elemento añadido a la pila es el primero que se quita de ella.

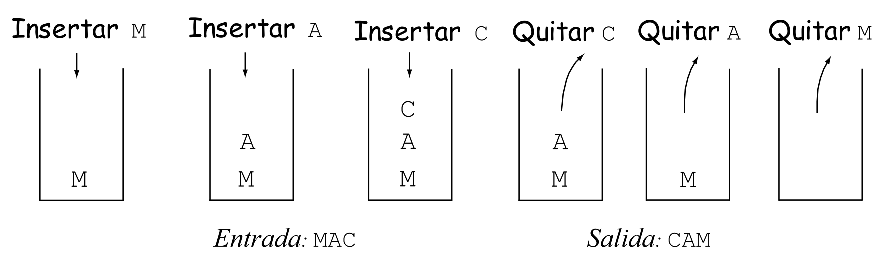


Figura 9.2 Poner y quitar elementos de la pila

La operación `Insertar` *(push)* sitúa un elemento dato en la cima de la pila, y `Quitar` *(pop)* elimina o quita el elemento de la pila.

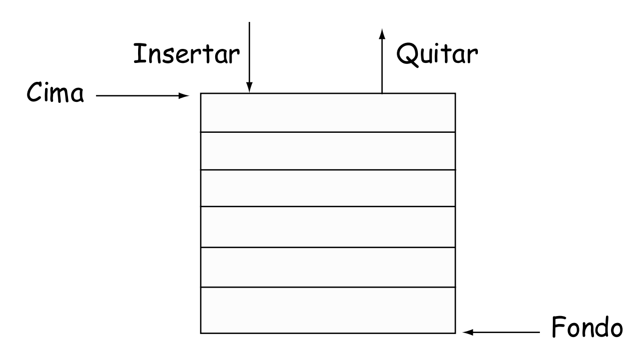


Figura 9.3 Operaciones básicas de una pila

La pila se puede implementar guardando los elementos en un *array,* en cuyo caso su dimensión o longitud es fija. También se puede utilizar un `Vector` para almacenar los elementos. Otra forma de implementación consiste en construir una lista enlazada, de modo que cada elemento de la pila forma un nodo de la lista. La lista crece o decrece según se añaden o se extraen, respectivamente, elementos de la pila; ésta es una representación dinámica, y no existe limitación en su tamaño excepto la memoria del computador.

Una pila puede estar *vacía* (no tiene elementos) o *llena* (en la representación con un *array* —arreglo—, si se ha llegado al último elemento). Si un programa intenta sacar un elemento de una pila vacía, se producirá un error, una *excepción*, debido a que esa operación es imposible; esta situación se denomina **desbordamiento negativo** *(underflow).* Por el contrario, si un programa intenta poner un elemento en una pila *llena,* se produce un error, una *excepción*, de **desbordamiento** (*overflow*) o *rebosamiento.* Para evitar estas situaciones se diseñan métodos que comprueban si la pila está llena o vacía.

9.1.1. Especificaciones de una pila

Las operaciones que sirven para definir una pila y poder manipular su contenido son las siguientes (no todas ellas se implementan al definir una pila):

Tipo de dato	Dato que se almacena en la pila.
Operaciones	
Crear Pila	Inicia.
Insertar (push)	Pone un dato en la pila.
Quitar (pop)	Retira (saca) un dato de la pila.
Pila vacía	Comprueba si la pila no tiene elementos.
Pila llena	Comprueba si la pila está llena de elementos.
Limpiar pila	Quita todos sus elementos y deja la pila vacía.
Cima Pila	Obtiene el elemento cima de la pila.
Tamaño de la pila	Número de elementos máximo que puede contener la pila.

9.2. TIPO DE DATO PILA IMPLEMENTADO CON *ARRAYS*

Los elementos que forman la pila se guardan en *arrays* (arreglos), en el contenedor `Vector` o bien formando una lista enlazada. La implementación con un *array* (arreglo) es *estática* ya que el *arrray* es de tamaño fijo.

La clase `PilaLineal`, con esta representación, necesita un *array* y una variable numérica, `cima`, que apunte al último elemento colocado en la pila. Al utilizar un *array* es necesario tener en cuenta que el tamaño de la pila no puede exceder el número de elementos del *array,* y la condición *pila llena* será significativa para el diseño.

El método usual de introducir elementos en una pila es definir el *fondo* de la pila en la posición `0` del *array* y sin ningún elemento en su interior, es decir, definir una *pila vacía*; a continuación, se van introduciendo elementos en la pila (en el *array*), de modo que el primer elemento añadido se introduce en una pila vacía y en la posición `0`, el segundo elemento en la posición `1`, el siguiente en la posición `2` y así sucesivamente. Con estas operaciones, el índice que apunta a la cima de la pila se va incrementando en `1` cada vez que se añade un nuevo elemento. Los algoritmos de introducir, "insertar" *(push)* y "quitar", sacar, *(pop)* datos de la pila son:

```
Insertar (push)

1. Verificar si la pila no está llena.
2. Incrementar en 1 el puntero índice de la pila.
3. Almacenar elemento en la posición del puntero de la pila.
```

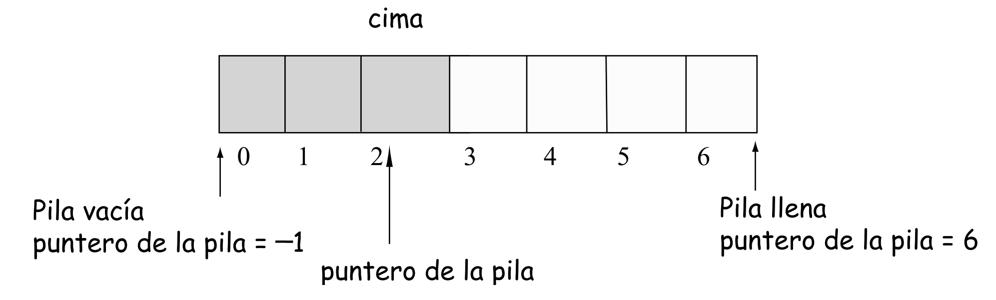
```
Quitar (pop)

1. Verificar si la pila no está vacía.
2. Leer el elemento de la posición del puntero de la pila.
3. Decrementar en 1 el puntero de la pila.
```

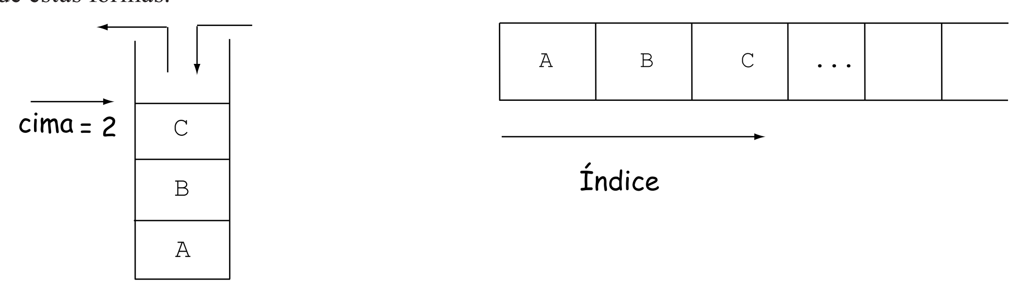
El rango de elementos que puede tener una pila varía de `0` a `TAMPILA-1` en el supuesto de que el *array* se defina de tamaño `TAMPILA` elementos. De modo que, *en una pila llena,* el puntero (índice del *array*) de la pila tiene el valor `TAMPILA-1,` y *en una pila vacía* el puntero tendrá el valor `-1` (el valor `0`, teóricamente, es el índice del primer elemento).

Ejemplo 9.1

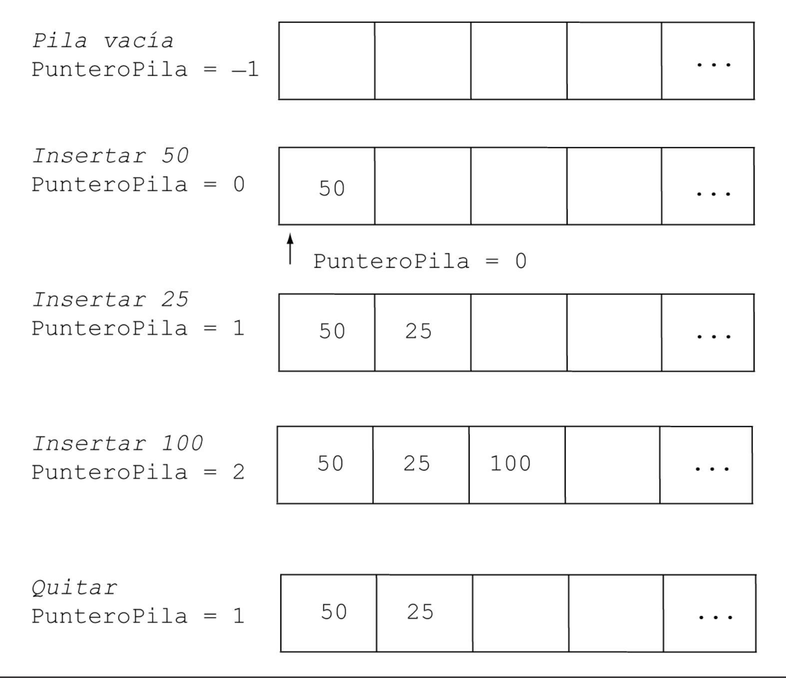
Una pila de 7 elementos se puede representar gráficamente así:



Si se almacenan los datos `A, B, C`, ... en la pila se puede representar gráficamente de alguna de estas formas:



A continuación se muestra la imagen de una pila según diferentes operaciones realizadas en un posible programa.



9.2.1. Clase PilaLineal

La declaración de un tipo abstracto incluye la representación de los datos y la definición de las operaciones. En el *TAD* Pila los datos pueden ser de cualquier tipo, y las operaciones, las citadas anteriormente en el apartado 9.1.1.

1. Datos de la pila (TipoDato es cualquier tipo de dato primitivo o tipo clase).
2. crearPila inicializa una pila. Es como crear una pila sin elementos, por tanto, vacía.
3. Verificar que la pila no está llena antes de insertar o poner ("*push*") un elemento en la pila; verificar que una pila no está vacía antes de quitar o sacar ("*pop*") un elemento de la pila. Si estas precondiciones no se cumplen, se debe visualizar un mensaje de error y el programa debe terminar.
4. pilaVacia devuelve *verdadero* si la pila está vacía y *falso* en caso contrario.
5. pilaLlena devuelve *verdadero* si la pila está llena y *falso* en caso contrario. Estas dos últimas operaciones se utilizan para verificar las precondiciones de *insertar* y *quitar*.
6. limpiarPila vacía la pila, dejándola sin elementos y disponible para otras tareas.
7. cimaPila devuelve el valor situado en la cima de la pila, pero no se decrementa su puntero, ya que la pila queda intacta.

Definición

```
package TipoPila;

public class PilaLineal
{
   private static final int TAMPILA = 49;
   private int cima;
   private TipoDeDato []listaPila;

   public PilaLineal()
   {
      cima = -1; // condición de pila vacía
      listaPila = new TipoDeDato[TAMPILA];
   }
   // operaciones que modifican la pila
   public void insertar(TipoDeDato elemento){...}
   public TipoDeDato quitar(){...}
   public void limpiarPila(){...}
   // operación de acceso a la pila
   public  TipoDeDato cimaPila(){...}
   // verificación estado de la pila
   public boolean pilaVacia(){...}
   public boolean pilaLlena(){...}
}
```

Ejemplo 9.2

Escribir un programa que cree una pila de enteros y realice operaciones de añadir datos a la pila, quitar...

Se supone implementada la pila con el tipo primitivo int. El programa crea una pila de números enteros, inserta en la pila elementos leídos del teclado (hasta leer la clave -1) y a

continuación extrae los elementos de la pila hasta que se vacía. El bloque de sentencias se encierra en un bloque `try` para tratar `errores` de `desbordamiento` de la pila.

```
import TipoPila.PilaLineal;
import java.io.*;

class EjemploPila
{
   public static void main(String [] a)
   {
      PilaLineal pila;
      int x;
      final int CLAVE = -1;
      BufferedReader entrada = new BufferedReader(
                          new InputStreamReader(System.in));

      System.out.println("Teclea los elemento (termina con -1).");
      try {
         pila = new PilaLineal(); // crea pila vacía
         do {
            x = Integer.parseInt(entrada.readLine());
            pila.insertar(x);
         }while (x != CLAVE);

         System.out.println("Elementos de la Pila: ");
         while (!pila.pilaVacia())
         {
            x = pila.quitar();
            System.out.print(x + " ");
         }
      }
      catch (Exception er)
      {
         System.err.println("Excepcion: " + er);
      }
   }
}
```

La declaración realizada está ligada al tipo de los elementos de la pila. Para alcanzar la máxima abstracción, se puede declarar la clase `PilaLineal` de tal forma que `TipoDeDato` sea el tipo referencia `Object`. Ésta es la clase base de todas las clases de Java y, por tanto, hay una conversión automática de cualquier referencia a `Object`. Como contrapartida, se necesita, en las operaciones que extraen elementos, convertir el tipo `Object` al tipo concreto de elemento. Además, cuando se necesite una pila de elementos de tipo primitivo, por ejemplo `int`, se tiene que crear una referencia que *envuelva* al elemento `(Integer, Double, ... )`.

Ejemplo 9.3

Declarar la clase `Pila` *con elementos de tipo* `Object`*. Insertar y extraer de la pila datos de tipo entero.*

La declaración de la clase:

```
package TipoPila;

public class PilaLineal
```

```
{
   private static final int TAMPILA = 49;
   private int cima;
   private Object [] listaPila;
      // operaciones que añaden o extraen elementos
   public void insertar(Object elemento)
   public Object quitar()throws Exception
   public  Object cimaPila()throws Exception
     // resto de operaciones
   }
```

El siguiente segmento de código inserta los datos `11, 50 y 22`:

```
PilaLineal pila = new PilaLineal();
pila.insertar(new Integer(11));
pila.insertar(new Integer(50));
pila.insertar(new Integer(22));
```

Para extraer de la pila y asignar el dato a una variable:

```
Integer dato;
dato = (Integer) pila.quitar();
```

9.2.2. Implementación de las operaciones sobre pilas

Los métodos de la clase `Pila` se implementan fácilmente, teniendo en cuenta la característica principal de esta estructura: *inserciones y borrados se realizan por el mismo extremo, la cima de la pila*.

Las operaciones `insertar()` y `quitar()` añaden y eliminan, respectivamente, un elemento de la pila; la operación `cimaPila` permite a un cliente recuperar los datos de la cima de la pila sin quitar realmente el elemento de la misma.

La operación de `insertar` un elemento en la pila incrementa el puntero `cima` de la pila en 1 y asigna el nuevo elemento a la lista. Cualquier intento de añadir un elemento en una pila llena genera una excepción o error debido al *desbordamiento de la pila*.

```
public void insertar(TipoDeDato elemento)throws Exception
{
   if (pilaLlena())
   {
      throw new Exception("Desbordamiento pila");
   }
      //incrementar puntero cima y copia elemento
   cima++;
   listaPila[cima] = elemento;
}
```

La operación `quitar` elimina un elemento de la pila copiando, primero, el valor de la cima de la pila en una variable local, `aux` y, a continuación, decrementando el puntero de la pila en 1. El método `quitar` devuelve la variable `aux`, es decir el elemento eliminado por la operación. Si se intenta eliminar o borrar un elemento en una pila vacía se produce error, se lanza una excepción general.

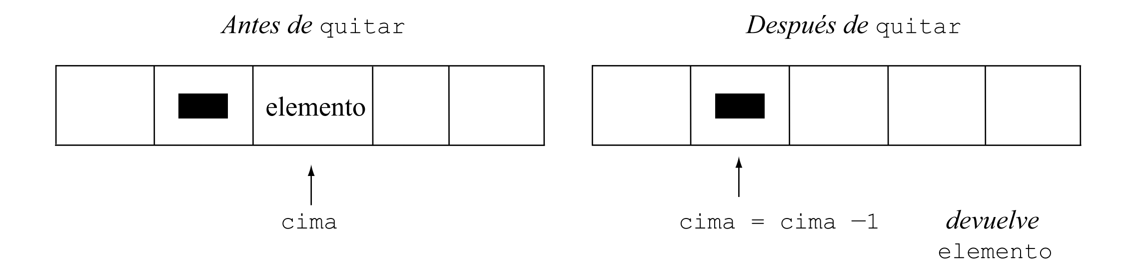


Figura 9.4 Extraer elemento `cima`

```
public TipoDeDato quitar() throws Exception
{
   TipoDeDato aux;
   if (pilaVacia())
   {
      throw new Exception ("Pila vacía, no se puede extraer.");
   }
      // guarda elemento de la cima
   aux = listaPila[cima];
      // decrementar cima y devolver elemento
   cima--;
   return aux;
}
```

La operación `cimaPila` devuelve el elemento que se encuentra en la cima de la pila. No se modifica la pila, únicamente se accede al elemento.

```
public TipoDato cimaPila() throws Exception
{
   if (pilaVacia())
   {
      throw new Exception ("Pila vacía, no hay elementos.");
   }
   return listaPila[cima];
}
```

9.2.3. Operaciones de verificación del estado de la pila

Se debe proteger la integridad de la pila, para lo cual el tipo `Pila` ha de proporcionar operaciones que comprueben su estado: *pila vacía* o *pila llena*. Asimismo, se ha de definir una operación, `LimpiarPila`, que restaure la condición inicial de la pila, que fue determinada por el constructor `Pila` (cima de la pila a `-1`).

El método `pilaVacia` comprueba (verifica) si la cima de la pila es `-1`. En ese caso, la pila está vacía y devuelve *verdadero*; en caso contrario, devuelve *falso*.

```
public boolean pilaVacia()
{
   return cima == -1;
}
```

El método `pilaLlena` comprueba (verifica) si la cima es `TAMPILA - 1`. En ese caso, la pila está llena y devuelve *verdadero;* en caso contrario, devuelve *falso*.

```
public boolean pilaLlena()
{
   return cima == TAMPILA - 1;
}
```

Por último, la operación `limpiarPila` pone la cima de la pila a su valor inicial.

```
public void limpiarPila()
{
   cima = -1;
}
```

Ejercicio 9.1

Escribir un programa que utilice una `Pila` *para comprobar si una determinada frase/palabra (cadena de caracteres) es un palíndromo.* ***Nota:*** *una palabra o frase es un palíndromo cuando la lectura directa e indirecta de la misma tiene igual valor:* ***alila,*** *es un palíndromo;* ***cara (arac)*** *no es un palíndromo.*

La palabra se lee con el método `readLine()` y se almacena en un `String`; cada carácter de la palabra leída se inserta en una pila de caracteres. Una vez leída la palabra y construida la pila, se compara el primer carácter del `String` con el carácter que se extrae de la pila; si son iguales, sigue la comparación con el siguiente carácter del `String` y de la pila; así sucesivamente hasta que la pila se queda vacía o hay un carácter no coincidente. Al guardar los caracteres de la palabra en la pila se garantiza que las comparaciones de caracteres se realizan en orden inverso: primero con último…

Se codifica de nuevo la clase `Pila`, cambiando el tipo de los elementos a `Object`. Para ello es necesario crear para cada carácter (tipo `char`) una referencia a la clase `Character`.

```
import java.io.*;

class PilaLineal
{
   private static final int TAMPILA = 79;
   private int cima;
   private Object [] listaPila;

   public PilaLineal()
   {
      cima = -1;
      listaPila = new Object[TAMPILA];
   }
   public void insertar(Object elemento) throws Exception
   {
      if (pilaLlena())
      {
         throw new Exception("Desbordamiento pila");
      }
      cima++;
```

```
      listaPila[cima] = elemento;
   }
   public Object quitar() throws Exception
   {
      Object aux;
      if (pilaVacia())
      {
         throw new Exception ("Pila vacía, no se puede extraer.");
      }
      aux = listaPila[cima];
      cima--;
      return aux;
   }
   public Object cimaPila() throws Exception
   {
      if (pilaVacia())
      {
         throw new Exception ("Pila vacía, no se puede extraer.");
      }
      return listaPila[cima];
   }
   public boolean pilaVacia()
   {
      return cima == -1;
   }
   public boolean pilaLlena()
   {
      return cima == TAMPILA-1;
   }
   public void limpiarPila()
   {
      cima = -1;
   }

}

public class Palindromo
{
   public static void main(String [] a)
   {
      PilaLineal pilaChar;
      char ch;
      boolean esPal;
      String pal;
      BufferedReader entrada = new BufferedReader(
                           new InputStreamReader(System.in));
      try {
         pilaChar = new PilaLineal(); // crea pila vacía
         System.out.print("Teclea la palabra" +
                           " a verificar si es palíndromo: ");
         pal = entrada.readLine();
            // se crea la pila con los caracteres de la palabra
         for (int i = 0; i < pal.length(); )
         {
            Character c;
            c = new Character(pal.charAt(i++));
            pilaChar.insertar(c);
```

```
         }
            // se comprueba si es palíndromo
         esPal = true;
         for (int j = 0; esPal && !pilaChar.pilaVacia(); )
         {
            Character c;
            c = (Character) pilaChar.quitar();
            esPal = pal.charAt(j++) == c.charValue();
         }

         pilaChar.limpiarPila();
         if (esPal)
            System.out.println("La palabra " + pal +
                               " es un palíndromo \n");
         else
            System.out.println("La palabra " + pal +
                               " no es un palíndromo \n");
      }
      catch (Exception er)
      {
         System.err.println("Excepcion: " + er);
      }
   }

}
```

9.3. PILA DINÁMICA IMPLEMENTADA CON UN VECTOR

La clase `Vector` es un contenedor de objetos que puede ser manejado para que crezca y decrezca dinámicamente. Los elementos de `Vector` son de tipo `Object`. Dispone de métodos para asignar un elemento en una posición (`insertElementAt`), para añadir un elemento a continuación del último (`addElement`), para obtener el elemento que se encuentra en una posición determinada (`elementAt`), para eliminar un elemento (`removeElementAt`),... Entonces, es posible mejorar la implementación del tipo abstracto `Pila` utilizando como depósito de los elementos un objeto `Vector`.

La posición del último elemento añadido a la pila se mantiene con la variable `cima`. Inicialmente, `cima = -1`, que es la condición de *pila vacía*.

Insertar un nuevo elemento supone aumentar `cima` y asignar el elemento a la posición `cima` del `Vector`. La operación se realiza llamando al método `addElement()`, que añade un elemento a partir del último y, si es necesario, aumenta la capacidad. No es necesario implementar el método `pilaLlena()` ya que la capacidad del `Vector` crece dinámicamente.

El método `quitar()` devuelve el elemento `cima` de la pila y lo elimina. Al utilizar un `Vector`, llamando a `removeElementAt(cima)` se elimina el elemento `cima`, y a continuación se decrementa `cima`.

A continuación se implementa la clase `PilaVector`. El constructor crea un `Vector` con una capacidad inicial e inicializa `cima` a `-1`.

```
package TipoPila;
import java.util.Vector;

public class PilaVector
```

```
{
   private static final int INICIAL = 19;
   private int cima;
   private Vector listaPila;

   public PilaVector()
   {
      cima = -1;
      listaPila = new Vector(INICIAL);
   }
   public void insertar(Object elemento)throws Exception
   {
      cima++;
      listaPila.addElement(elemento);
   }
   public Object quitar() throws Exception
   {
      Object aux;
      if (pilaVacia())
      {
         throw new Exception ("Pila vacía, no se puede extraer.");
      }
      aux = listaPila.elementAt(cima);
      listaPila.removeElementAt(cima);
      cima--;
      return aux;
   }
   public Object cimaPila() throws Exception
   {
      if (pilaVacia())
      {
         throw new Exception ("Pila vacía, no se puede extraer.");
      }
      return listaPila.elementAt(cima);
   }
   public boolean pilaVacia()
   {
      return cima == -1;
   }
   public void limpiarPila()throws Exception
   {
      while (! pilaVacia())
         quitar();
   }

}
```

Nota de programación

Para utilizar una pila de elementos de tipo primitivo (`int`, `char`, `long`, `float`, `double...`) es necesario, para insertar, crear un objeto de la correspondiente clase *envolvente* (`Integer`, `Character`, `Long`, `Float`, `Double...`) y pasar dicho objeto como argumento del método `insertar()`.

Ejemplo 9.4

Llenar una pila con números leídos del teclado. A continuación, vaciar la pila de tal forma que se muestren los valores positivos.

Se utiliza la declaración de `Pila` del apartado 9.3. El número leído del teclado se convierte a una referencia a `Double` y se inserta en la pila. Para vaciar la pila se diseña un bucle, *hasta pila vacía* y cada elemento que se extrae se escribe si es positivo.

```
import java.io.*;
import TipoPila.*;

class EjemploPilaDinamica
{
   public static void main(String [] a)
   {
      PilaVector pila = new PilaVector();
      int x;
      BufferedReader entrada = new BufferedReader(
                              new InputStreamReader(System.in));

      System.out.println("Teclea número de elementos: ");
      try {
         x = Integer.parseInt(entrada.readLine());
         for (int j = 1; j <= x; j++)
         {
            Double d;
            d = Double.valueOf(entrada.readLine());
            pila.insertar(d);
         }
            // vaciado de la pila
         System.out.println("Elementos de la Pila: ");
         while (!pila.pilaVacia())
         {
            Double d;
            d = (Double)pila.quitar();
            if (d.doubleValue() > 0.0)
               System.out.print(d + " ");
         }
      }
      catch (Exception er)
      {
         System.err.println("Excepcion: " + er);
      }
   }
}
```

9.4. EL TIPO PILA IMPLEMENTADO COMO UNA LISTA ENLAZADA

La realización dinámica de una pila utilizando una lista enlazada almacena cada elemento de la pila como un nodo de la lista, con la particularidad típica del *TAD* `Pila`: se inserta un elemento por el mismo extremo por el que se extrae, es decir, por la cima de la pila.

Esta realización tiene la ventaja de que el tamaño se ajusta exactamente al número de elementos de la pila. Sin embargo, para cada elemento es necesaria más memoria ya que hay que guardar el campo de enlace entre nodos consecutivos. La Figura 9.5 muestra la imagen de una pila implementada con una lista enlazada.

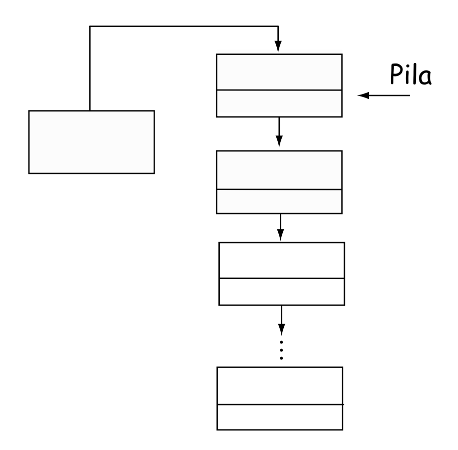


Figura 9.5 Representación de una pila con una lista enlazada

9.4.1. Clase `Pila` y `NodoPila`

La estructura que tiene la pila implementada con una lista enlazada es muy similar a la expuesta en listas enlazadas. Los elementos de la pila son los nodos de la lista, con un campo para guardar el elemento y otro de enlace. Las operaciones del tipo pila implementada con listas son, naturalmente, las mismas que si la pila se implementa con *arrays,* salvo la operación que controla si la pila está llena, `pilaLlena`, que ahora no tiene significado ya que la listas enlazadas crecen indefinidamente, con el único límite de la memoria.

La clase `NodoPila` representa un nodo de la lista enlazada. Tiene dos atributos: `elemento` guarda el elemento de la pila y `siguiente` contiene la dirección del siguiente nodo de la lista. El constructor pone el dato en `elemento` e inicializa `siguiente` a `null`. El tipo de dato de `elemento` se corresponde con el tipo de los elementos de la pila para que no dependa de un tipo concreto; para que sea más genérico, se utiliza el tipo `Object` y, de esa forma, puede almacenar cualquier tipo de referencia.

```
package TipoPila;

public class NodoPila
{
   Object elemento;
   NodoPila siguiente;

   NodoPila(Object x)
   {
      elemento = x;
      siguiente = null;
   }
}
```

La clase `PilaLista` implementa las operaciones del *TAD* `Pila`. Además, dispone del atributo `cima` que es la dirección (referencia) al primer nodo de la lista. El constructor inicializa la pila vacía (`cima = null`), realmente, a la condición de *lista vacía*.

```
package TipoPila;

public class PilaLista
{
   private NodoPila cima;

   public PilaLista()
   {
      cima = null;
   }
      // operaciones
}
```

9.4.2. Implementación de las operaciones del TAD Pila con listas enlazadas

Las operaciones `insertar`, `quitar`, `cima`, acceden a la lista directamente con la referencia `cima` (apunta al último nodo apilado). Entonces, como no necesitan recorren los nodos de la lista, no dependen del número de nodos, la eficiencia de cada operación es constante, `O(1)`.

La codificación que a continuación se escribe es para una pila de elemento de cualquier tipo. Es preciso recordar que, al quitar el elemento cima, será necesario convertir el tipo `Object` al *tipo clase* de los elementos actuales. La clase `PilaLista` forma parte del mismo paquete que `NodoLista`, por lo que tiene acceso a todos sus miembros.

Verificación del estado de la pila.

```
public boolean pilaVacia()
{
   return cima == null;
}
```

Poner un elemento en la pila. Crea un nuevo nodo con el elemento que se pone en la pila y se enlaza por la cima.

```
public void insertar(Object elemento)
{
   NodoPila nuevo;
   nuevo = new NodoPila(elemento);
   nuevo.siguiente = cima;
   cima = nuevo;
}
```


Figura 9.6 Apilar un elemento

Eliminación del elemento cima. Retorna el elemento cima y lo quita de la pila.

```
public Object quitar() throws Exception
{
   if (pilaVacia())
      throw new Exception ("Pila vacía, no se puede extraer.");

   Object  aux = cima.elemento;
   cima = cima.siguiente;
   return aux;
}
```

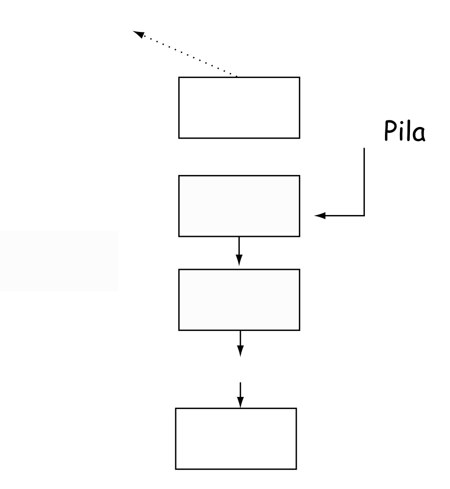


Figura 9.7 Quita la cima de la pila

Obtención del elemento cabeza o cima de la pila, sin modificar la pila:

```
public Object cimaPila() throws Exception
{
   if (pilaVacia())
      throw new Exception ("Pila vacía, no se puede leer cima.");

   return cima.elemento;
}
```

Vaciado de la pila. Libera todos los nodos de que consta la pila. Recorre los `n` nodos de la lista enlazada; por consiguiente, es una operación lineal, `O(n)`.

```
public void limpiarPila()
{
   NodoPila t;
   while(!pilaVacia())
   {
      t = cima;
      cima = cima.siguiente;
      t.siguiente = null;
   }
   }
```

9.5. EVALUACIÓN DE EXPRESIONES ARITMÉTICAS CON PILAS

Una *expresión aritmética* está formada por operandos y operadores. La expresión `x*y - (a+b)` consta de los operadores `*, -, +` y de los operandos `x, y, a, b`. Los operandos pueden ser valores constantes, variables o, incluso, otra expresión. Los operadores son los símbolos conocidos de las operaciones matemáticas. La evaluación de una expresión aritmética da lugar a un valor numérico, se realiza sustituyendo los operandos que son variables por valores concretos y ejecutando las operaciones aritméticas representadas por los operadores. Si los operandos de la expresión anterior toman los valores `x = 5, y = 2, a = 3, b = 4`, el resultado de la evaluación es:

```
5*2 - (3+4)  = 5*2 - 7 = 10 - 7 = 3
```

La forma habitual de escribir expresiones matemáticas es aquella en la que el operador está entre sus dos operandos. La expresión anterior está escrita de esa forma, y recibe el nombre de *notación infija*. Esta forma de escribir las expresiones exige, en algunas ocasiones, el uso de paréntesis para *encerrar* subespresiones con mayor prioridad.

Los operadores, como es sabido, tienen distintos niveles de precedencia o prioridad a la hora de su evaluación. A continuación se recuerdan estos niveles de prioridad en orden de mayor a menor:

```
Paréntesis               :  ( )
Potencia                 :   ^
Multiplicación/división  :   *, /
Suma/Resta               :   +, -
```

Normalmente, en una expresión hay operadores con la misma prioridad. A igualdad de precedencia, los operadores se evalúan de izquierda a derecha *(asociatividad)*, excepto la potencia, que es de derecha a izquierda.

9.5.1. Notación prefija y notación postfija de una expresiones aritmética

La forma habitual de escribir operaciones aritméticas *(notación infija)* sitúa el operador entre sus dos operandos. Esta forma de notación obliga, en muchas ocasiones, a utilizar paréntesis para indicar el orden de evaluación. Las expresiones siguientes:

```
r =  a*b/(a+c)      g = a*b/a+c
```

son distintas al no poner paréntesis en la expresión `g`. Igual ocurre con estas otras:

```
r =  (a-b)^c+d      g = a-b^c+d
```

Existen otras formas de escribir expresiones aritméticas, que se diferencian por la ubicación del operador respecto de los operandos. La notación en la cual el operador se coloca delante de los dos operandos, se conoce como *notación prefija* y se conoce como notación polaca (en honor del matemático polaco que la propuso).

Ejemplo 9.5

Dadas las expresiones: `a*b/(a+c);` `a*b/a+c;` `(a-b)^c+d` *. Escribir las expresiones equivalentes en notación prefija.*

Se escribe, paso a paso, la transformación de cada expresión aritmética en la expresión equivalente en notación polaca.

```
a*b/(a+c)   (infija) → a*b/+ac  → *ab/+ac  → /*ab+ac (polaca)
a*b/a+c     (infija) → *ab/a+c  → /*aba+c  → +/*abac (polaca)
(a-b)^c+d   (infija) → -ab^c+d → ^-abc+d  → +^-abcd (polaca)
```

En el Ejemplo 9.4, se observa que no son necesarios los paréntesis al escribir la expresión en notación polaca. Ésta es su propiedad fundamental: el orden de ejecución de las operaciones está determinado por las posiciones de los operadores y los operandos en la expresión.

Notación postfija

Hay más formas de escribir las expresiones. La tercera notación más utilizada se denomina *postfija* o *polaca inversa* y coloca el operador a continuación de sus dos operandos.

Ejemplo 9.6

Dadas las expresiones: `a*b/(a+c);` `a*b/a+c;` `(a-b)^c+d`*, escribir las expresiones equivalentes en notación postfija.*

Paso a paso, se transforma cada subexpresión en notación *postfija*.

```
a*b/(a+c)   (infija) → a*b/ac+  → ab*/ac+  → ab*ac+/  (polaca inversa)
a*b/a+c     (infija) → ab*/a+c  → ab*a/+c  → ab*a/c+  (polaca inversa)
(a-b)^c+d  (infija) → ab-^c+d  → ab-c^+d → ab-c^d+  (polaca inversa)
```

Norma

Las diferentes formas de escribir una misma expresión aritmética dependen de la ubicación de los operadores respecto a los operandos. Es importante tener en cuenta que tanto en la notación prefija como en la postfija no son necesarios los paréntesis para cambiar el orden de evaluación.

9.5.2. Evaluación de una expresión aritmética

La evaluación de una expresión aritmética escrita de *manera habitual,* en *notación infija,* se realiza en dos pasos principales:

1. Transformar la expresión de notación *infija* a *postfija*.
2. Evaluar la expresión en notación *postfija*.

En los algoritmos que se aplican en cada uno de los pasos es básico el uso del *TAD Pila*. La estructura pila asegura que el *último en entrar es el primero en salir* y, de esa forma, el algoritmo de transformación a *postfija* sitúa los operadores después de sus operandos con la prioridad o precedencia que les corresponde. Una vez que se tiene la expresión en notación *postfija,* se utiliza otra pila, de elementos numéricos, para guardar los valores de los operandos y de las operaciones parciales con el fin de obtener el valor numérico de la expresión.

9.5.3. Transformación de una expresión infija a postfija

Se parte de una expresión en notación *infija* que tiene operandos, operadores y puede tener paréntesis. Los operandos vienen representados por letras, mientras los operadores son éstos:

```
^ (potenciación), *, /, +, - .
```

La transformación se realiza utilizando una pila en la que se almacenan los operadores y los paréntesis izquierdos. La expresión aritmética se lee del teclado y se procesa carácter a carácter. Los operandos pasan directamente a formar parte de la expresión en *postfija,* la cual se guarda en un *array.* Un operador se mete en la pila si se cumple que:

- La pila esta vacía.
- El operador tiene mayor prioridad que el operador cima de la pila.
- El operador tiene igual prioridad que el operador cima de la pila y se trata de la máxima prioridad.

Si la prioridad es menor o igual, se saca el elemento cima de la pila, se pone la expresión en *postfija* y se vuelve a hacer la comparación con el nuevo elemento cima.

El paréntesis izquierdo siempre se mete en la pila; ya en la pila, se les considera de mínima prioridad para que todo operador que se encuentra dentro del paréntesis entre en la pila. Cuando se lee un paréntesis derecho, hay que sacar todos los operadores de la pila y ponerlos en la expresión *postfija,* hasta llegar a un paréntesis izquierdo, el cual se elimina, ya que los paréntesis no forman parte de la expresión *postfija*.

El algoritmo termina cuando no hay más ítems de la expresión origen y la pila está vacía. Por ejemplo, la expresión `a*(b+c-(d/e^f)-g)-h` está escrita en *notación infija*, la expresión equivalente en *postfija* se va a ir formando paso a paso:

Expresión en postfija

`a`	Operando `a` pasa a la expresión *postfija;* operador `*` a la pila.
`ab`	Operador `(` pasa a la pila; operando `b` a la expresión.
`abc`	Operador + pasa a la pila; operando `c` a la expresión.
	En este punto, el estado de la pila es:

```
| + |
| ( |
| * |
```

El siguiente carácter de la expresión, –, tiene igual prioridad que el operador de la cima (+) y da lugar a:

```
| - |
| ( |
| * |
+---+
```

`abc+`	
`abc+d`	El operador `(` se mete en la pila; el operando `d` a la expresión.
`abc+de`	El operador `/` pasa a la pila; el operando `e` a la expresión.
`abc+def`	El operador `^` pasa a la pila; el operando `f` a la expresión.
	El siguiente ítem, `)` (paréntesis derecho), produce que se vacíe la pila hasta un `'('`. La pila, en este momento, dispone de estos operadores:

```
| ^ |
| / |
| ( |
| - |
| ( |
| * |
+---+
```

`abc+def^/`	El algoritmo saca operadores de la pila hasta un `'('` y da lugar a la pila:

```
| - |
| ( |
| * |
+---+
```

`abc+def^/-`	El operador – pasa a la pila y, a su vez, se extrae –; el siguiente carácter, el operando `g`, pasa a la expresión.
`abc+def^/-g`	El siguiente carácter es `')'`, por lo que son extraídos de la pila los operadores hasta un `(`, la pila queda :

```
| * |
+---+
```

`abc+def^/-g-`	El siguiente carácter es el operador `–`, hace que se saque de la pila el operador `*` y se meta en la pila el operador –.
`abc+def^/-g-*`	Por último, el operando h pasa directamente a la expresión.
`abc+def^/-g-*h`	Fin de entrada, se vacía la pila pasando los operadores a la expresión:
`abc+def^/-g-*h-`	

En la descripción realizada se observa que el paréntesis izquierdo tiene la máxima prioridad fuera de la pila; es decir, en la *notación infija*; sin embargo, cuando está dentro de la pila, la prioridad es mínima. De igual forma, para tratar el hecho de que dos, o más, operadores de potenciación se evalúan de derecha a izquierda, este operador tendrá mayor prioridad cuando todavía no esté metido en la pila que cuando esté puesto en ella. Las prioridades son determinadas según la Tabla 9.1.

Tabla 9.1 Prioridades de los operadores considerados

Operador	Prioridad dentro pila	Prioridad fuera pila
`^`	3	4
`*, /`	2	2
`+,-`	1	1
`(`	0	5

Algoritmo de paso de notación *infija* a *postfija*

Los pasos a seguir para transformar una expresión algebraica de notación infija a postfija son:

1. Obtener caracteres de la expresión y repetir los pasos 2 al 4 para cada carácter.
2. Si es un operando, pasarlo a la expresión *postfija*.
3. Si es operador:
   3.1. Si la pila está vacía, meterlo en la pila. Repetir a partir de 1.
   3.2. Si la pila no está vacía:
   Si la prioridad del operador leído es mayor que la prioridad del operador cima de la pila, meterlo en la pila y repetir a partir de 1.
   Si la prioridad del operador es menor o igual que la prioridad del operador de la cima de la pila, sacar el operador cima de la pila y pasarlo a la expresión postfija, volver a 3.
4. Si es paréntesis derecho:
   4.1. Sacar el operador cima y pasarlo a la expresión postfija.
   4.2. Si el nuevo operador cima es paréntesis izquierdo, suprimir el elemento cima.
   4.3. Si la cima no es paréntesis izquierdo, volver a 4.1.
   4.4. Volver a partir de 1.
5. Si quedan elementos en la pila, pasarlos a la expresión postfija.
6. Fin del algoritmo.

Para codificar este algoritmo es necesario crear una pila en la que se almacenen caracteres. Se utiliza el diseño de pila con lista enlazada del apartado 9.4, que se encuentra en el paquete `TipoPila`. Es necesario, para poner un carácter en la pila, crear una referencia de tipo `Character`; al extraer o leer el elemento `cima`, se realizará la conversión inversa: de tipo `Object` a `Character` y llamando al método `charValue()` se obtiene el correspondiente carácter, que será un operador.

Codificación del algoritmo de transformación a postfija

En la clase `ExprePostfija`, se declaran los métodos para transformar una expresión escrita en *infija* a notación *postfija*. La clase hace de *depósito* de los métodos necesarios para realizar la transformación, todos ellos cualificados con `static`.

El método `postFija()` implementa los pasos del algoritmo de transformación, recibe como argumento una cadena con la expresión inicial, crea una pila y, en un bucle de tantas iteraciones como caracteres, realiza las acciones del algoritmo. Este método define el *array* `char[] expsion` de igual longitud que la expresión original. A este *array* se pasarán los elementos que forman la expresión en *postfija*. Una vez que termina la transformación, el método devuelve una cadena con la expresión escrita en notación *postfija;* la cadena se crea a partir del *array* `expsion: new String(expsion, 0, n)`.

El método `prdadDentro()` determina la prioridad de un operador que está dentro de la pila, y `prdadFuera()` la prioridad de un operador fuera de la pila.

(El código fuente se encuentra en la página web del libro, archivo *ExpresionPostfija*.)

9.5.4. Evaluación de la expresión en notación postfija

Una vez que se tiene la expresión en notación postfija, se evalúa la expresión. En primer lugar, hay que dar valores numéricos a los operandos de la expresión. De nuevo, el algoritmo de evaluación utiliza una pila, en esta ocasión de operandos, es decir, de números reales.

Al describir el algoritmo, `expsion` es la cadena con la expresión *postfija*. El número de elementos es la longitud, `n`, de la cadena. Los pasos a seguir son los siguientes:

1. Examinar `expsion` elemento a elemento: repetir los pasos 2 y 3 para cada elemento.
2. Si el elemento es un operando, meterlo en la pila.
3. Si el elemento es un operador, se simboliza con `&`, entonces:
   - Sacar los dos elementos superiores de la pila, que se denominarán `b` y `a` respectivamente.
   - Evaluar `a & b`, el resultado es `z = a & b`.
   - El resultado `z`, meterlo en la pila. Repetir a partir del paso 1.
4. El resultado de la evaluación de la expresión está en el elemento cima de la pila.
5. Fin del algoritmo.

Codificación de la evaluación de expresión en postfija

La clase `EvalPostfija` agrupa los métodos para implementar el algoritmo, todos ellos se declaran `static`. Realmente, son dos los métodos que se escriben: `valorOprdos()` y `evalua()`. El primero, `valorOprdos()`, simplemente es para dar entrada a los valores de los operandos.

El método `double evalua()` implementa el algoritmo y recibe la cadena con la expresión y el *array* con el valor de cada operando. La pila de números reales, utilizada por el algoritmo, se instancia de la clase `PilaLista`, diseñada de tal forma que sus elementos son de tipo `Object`. Entonces, al meter un operando en la pila será necesario crear instancias de tipo `Double`, y al extraer el elemento `cima`, convertir `Object` a `Double`.

(El código fuente se encuentra en la página web del libro, archivo *EvaluarExpresion*.)

Clase principal

El método `main()`gestiona la petición de la expresión original y las sucesivas llamadas a los métodos para transformar de *infija* a *postfija* y, por último, evaluar la expresión para unos valores concretos de los operandos.

```
import java.io.*;
import evaluarExpresion.*;

class EjemploEvaluaExpresion
{
   public static void main(String [] a)
   {
      double []v = new double[26];
      double resultado;
      BufferedReader entrada = new BufferedReader(
                              new InputStreamReader(System.in));
      String expresion;
      try {
         System.out.print("\nExpresión aritmética: ");
         expresion = entrada.readLine();
            // Conversión de infija a postfija
         expresion = ExprePostfija.postFija(expresion);
         System.out.println("\nExpresión en postfija: " + expresion);
            // Evaluación de la expresión
         EvalPostfija.valorOprdos(expresion, v);  // valor de operandos
         resultado = EvalPostfija.evalua(expresion, v);
         System.out.println("Resultado = " + resultado);
      }
      catch (Exception e)
      {
         System.out.println("\nError en el proceso ... " + e);
         e.getStackTrace();
      }
   }
}
```

RESUMEN

- Una *pila* es una estructura de datos tipo **LIFO** (*last in first out*, último en entrar primero en salir) en la que los datos (todos del mismo tipo) se añaden y se eliminan por el mismo extremo, denominado *cima* de la pila.
- Se definen las siguientes operaciones básicas sobre pilas: `crear`, `insertar`, `cima`, `quitar`, `pilaVacia`, `pilaLlena` y `limpiarPila`.
- `crear` inicializa la pila como pila vacía. Esta operación la implementa el constructor.
- `insertar` añade un elemento en la cima de la pila. Debe haber espacio en la pila.
- `cima` devuelve el elemento que está en la cima, sin extraerlo.
- `quitar` extrae de la pila el elemento cima de la pila.
- `pilaVacia` determina si el estado de la pila es vacía, en cuyo caso devuelve el valor lógico *true*.
- `pilaLlena` determina si existe espacio en la pila para añadir un nuevo elemento. De no haber espacio, devuelve `true`. Esta operación se aplica en la representación de la pila mediante *array*.

- `limpiarPila` el espacio asignado a la pila se libera, quedando disponible.
- Las aplicaciones de las pilas en la programación son numerosas; entre ellas está la evaluación de expresiones aritméticas. Primero se transforma la expresión a notación postfija, a continuación se evalúa.
- Las expresiones en notación polaca, postfija o prefija, tienen la característica de que no necesitan paréntesis.

EJERCICIOS

9.1. ¿Cuál es la salida de este segmento de código, teniendo en cuenta que el tipo de dato de la pila es `int`?

```
Pila p = new Pila();
int x = 4, y;

p.insertar(x);
System.out.println("\n " + p.cimaPila());
y = p.quitar();
p.insertar(32);
p.insertar(p.quitar());
do {
   System.out.println("\n " + p.quitar());
}while (!p.pilaVacia());
```

9.2. Escribir el método `mostarPila()` para escribir los elementos de una pila de cadenas de caracteres, utilizando sólo las operaciones básicas y una pila auxiliar.

9.3. Utilizando una pila de caracteres, transformar la siguiente expresión a su equivalente expresión en postfija.

`(x-y)/(z+w) – (z+y)^x`

9.4. Obtener una secuencia de 10 números reales, guardarlos en un *array* y ponerlos en una pila. Imprimir la secuencia original y, a continuación, imprimir la pila extrayendo los elementos.

9.5. Transformar la expresión aritmética del Ejercicio 9.3 en su expresión equivalente en notación prefija.

9.6. Dada la expresión aritmética `r = x*y - (z+w)/(z+y)^x`, transformar la expresión a notación postfija y, a continuación, evaluar la expresión para los valores: `x = 2, y = -1, z = 3, w = 1`. Para obtener el resultado de la evaluación seguir los pasos del algoritmo descrito en el Apartado 9.5.3.

9.7. Se tiene una lista enlazada a la que se accede por el primer nodo. Escribir un método que imprima los nodos de la lista en orden inverso, desde el último nodo al primero; como estructura auxiliar, utilizar una pila y sus operaciones.

9.8. La implementación del *TAD Pila* con *arrays* establece un tamaño máximo de la pila que se controla con el método `pilaLlena()`. Modificar este método de tal forma que, cuando se llene la pila, se amplíe el tamaño del *array* a justamente el doble de la capacidad actual.

PROBLEMAS

9.1. Escribir un método, `copiarPila()`, que copie el contenido de una pila en otra. El método tendrá dos argumentos de tipo pila, uno para la pila fuente y otro para la pila destino. Utilizar las operaciones definidas sobre el *TAD Pila*.

9.2. Escribir un método para determinar si una secuencia de caracteres de entrada es de la forma:

```
X  &  Y
```

siendo `X` una cadena de caracteres e `Y` la cadena inversa. El carácter `&` es el separador.

9.3. Escribir un programa que, haciendo uso de una Pila, procese cada uno de los caracteres de una expresión que viene dada en una línea. La finalidad es verificar el equilibrio de paréntesis, llaves y corchetes. Por ejemplo, la siguiente expresión tiene un número de paréntesis equilibrado:

```
((a+b)*5) - 7
```

A esta otra expresión le falta un corchete: `2*[(a+b)/2.5 + x - 7*y`

9.4. Escribir un programa en el que se manejen un total de $n = 5$ pilas: P_1, P_2, P_3, P_4 y P_5. La entrada de datos serán pares de enteros `(i,j)` tal que $1 \le abs(i) \le n$. De tal forma que el criterio de selección de pila será:

- Si `i` es positivo, debe insertarse el elemento `j` en la pila P_i.
- Si `i` es negativo, debe eliminarse el elemento `j` de la pila P_i.
- Si `i` es cero, fin del proceso de entrada.

Los datos de entrada se introducen por teclado. Cuando termina el proceso el programa debe escribir el contenido de la n Pilas en pantalla.

9.5. Modificar el programa del Problema 9.4 para que la entrada sean triplos de números enteros `(i,j,k)`, donde `i, j` tienen el mismo significado que en 9.4, y `k` es un número entero que puede tomar los valores `-1, 0` con este significado:

- `-1`, hay que borrar todos los elementos de la pila.
- `0`, el proceso es el indicado en el Problema 9.4 con `i` y `j`.

9.6. Se quieren determinar las frases que son palíndromo, para lo cual se ha de seguir la siguiente estrategia: considerar cada línea de una frase; añadir cada carácter de la frase a una pila y, a la vez, a lista enlazada circular por el final; extraer carácter a carácter, simultáneamente de la pila y de la lista circular el *primero,* su comparación determina si es palíndromo o no. Escribir un programa que lea líneas y determine si son palíndromo.

9.7. La función de Ackermann se define de la siguiente forma:

$A(m, n) = n + 1$ si $m = 0$

$A(m, n) = A(m - 1, 1)$ si $n = 0$

$A(m, n) = A(m - 1, A(m, n - 1))$ si $m > 0$, y $n > 0$

Se obseva que la definición es recursiva y, por consiguiente, la implementación recursiva se codifica de manera inmediata. Como alternativa, escribir un método que resuelva la función iterativamente utilizando el *TAD Pila*.

CAPITULO 10

Colas

Objetivos

Con el estudio de este capítulo, usted podrá:

- Especificar el tipo abstracto de datos Cola.
- Encontrar las diferencias fundamentales entre Pilas y Colas.
- Definir una clase Cola con *arrays.*
- Definir una clase Cola con listas enlazadas.
- Aplicar el tipo abstracto Cola para la resolución de problemas.

Contenido

Conceptos clave

- *Array circular*.
- Cola de objetos.
- Herencia de clase.
- Lista enlazada.
- Lista FIFO.
- Prioridad.

INTRODUCCIÓN

En este capítulo se estudia el tipo abstracto de datos *Cola,* estructura muy utilizada en la vida cotidiana y también en la resolución de problemas en programación. Esta estructura, al igual que las pilas, almacena y recupera sus elementos atendiendo a un orden estricto. Las colas se conocen como estructuras **FIFO** (*first-in, first-out,* primero en entrar-primero en salir), debido a la forma y orden de inserción y de extracción de elementos. Las colas tienen numerosas aplicaciones en el mundo de la computación: colas de mensajes, colas de tareas a realizar por una impresora, colas de prioridades, etc.

10.1. CONCEPTO DE COLA

Una **cola** es una estructura de datos que almacena elementos en una lista y permite acceder a los datos por uno de los dos extremos de la lista (Figura 10.1). Un elemento se inserta en la cola (parte *final*) de la lista y se suprime o elimina por el frente (parte inicial, *frente*) de la lista. Las aplicaciones utilizan una cola para almacenar elementos en su orden de aparición o concurrencia.

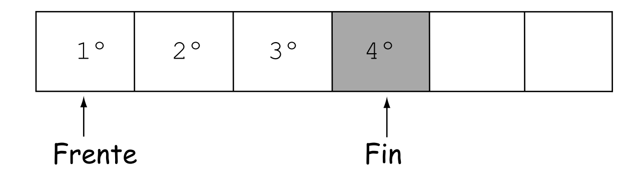


Figura 10.1 Una cola

Los elementos se eliminan (se quitan) de la cola en el mismo orden en que se almacenan y, por consiguiente, una cola es una estructura de tipo **FIFO** (*first-in, firs-out*, primero en entrar-primero en salir o bien primero en llegar-primero en ser servido). El servicio de atención a clientes en un almacén es un ejemplo típico de cola. La acción de gestión de memoria intermedia *(buffering)* de trabajos o tareas de impresora en un distribuidor de impresoras *(spooler)* es otro ejemplo típico de cola[1]. Dado que la impresión es una tarea (un trabajo) que requiere más tiempo que el proceso de la transmisión real de los datos desde la computadora a la impresora, se organiza una cola de trabajos de modo que los trabajos se imprimen en el mismo orden en el que se recibieron por la impresora. Este sistema tiene el gran inconveniente de que si su trabajo personal consta de una única página para imprimir y delante de su petición de impresión existe otra petición para imprimir un informe de 300 páginas, deberá esperar a la impresión de esas 300 páginas antes de que se imprima su página.

> ***A recordar***
>
> Una cola es una estructura de datos cuyos elementos mantienen un cierto orden, de tal modo que sólo se pueden añadir elementos por un extremo, **final** de la cola, y eliminar o extraer por el otro extremo, llamado **frente.**

---

[1] Recordemos que este caso sucede en sistemas multiusuario donde hay varios terminales y sólo una impresora de servicio. Los trabajos se "encolan" en la cola de impresión.

Las operaciones usuales en las colas son *Insertar* y *Quitar*. La operación **`Insertar`** añade un elemento por el extremo **final** de la cola, y la operación **`Quitar`** elimina o extrae un elemento por el extremo opuesto, el **frente** o primero de la cola. La organización de elementos en forma de cola asegura que *el primero en entrar es el primero en salir.*

10.1.1. Especificaciones del tipo abstracto de datos Cola

Las operaciones que sirven para definir una cola y poder manipular su contenido son las siguientes:

Tipo de dato Elemento que se almacena en la cola.

Operaciones

CrearCola	Inicia la cola como vacía.
Insertar	Añade un elemento por el final de la cola.
Quitar	Retira (extrae) el elemento frente de la cola.
Cola vacía	Comprueba si la cola no tiene elementos.
Cola llena	Comprueba si la cola está llena de elementos.
Frente	Obtiene el elemento frente o primero de la cola.
Tamaño de la cola	Número de elementos máximo que puede contener la cola

En una cola, al igual que en una pila, los datos se almacenan de un modo lineal y el acceso a los datos sólo está permitido en los extremos de la cola.

La forma que los lenguajes tienen para representar el *TAD* `Cola` depende de donde se almacenen los elementos: en un *array,* en una estructura dinámica como puede ser un `Vector` (contenedor de Java) o en una lista enlazada. La utilización de *arrays* tiene el problema de que la *cola* no puede crecer indefinidamente, está limitada por el tamaño del *array;* como contrapartida, el acceso a los extremos es muy eficiente. Utilizar una lista enlazada permite que el número de nodos se ajuste al de elementos de la cola; por el contrario, cada nodo necesita memoria extra para el enlace, y también hay que tener en cuenta el límite de memoria de la pila del computador.

x
Frente Fin
Insertar x

x y
Frente Fin
Insertar y

x y z
Frente Fin
Insertar z

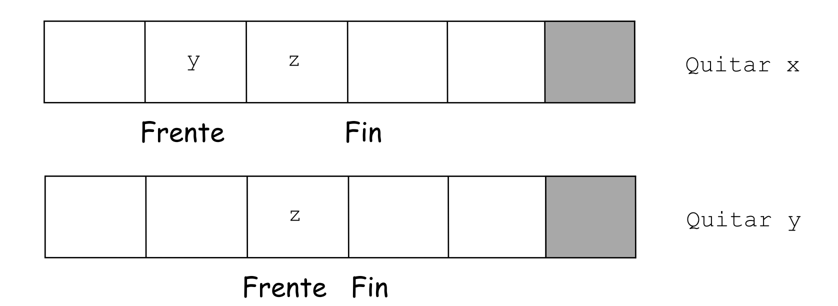


Figura 10.2 Operaciones *Insertar* y *Quitar* en una Cola

10.2. COLAS IMPLEMENTADAS CON *ARRAYS*

Al igual que las pilas, las colas se implementan utilizando una estructura estática *(arrays)* o una estructura dinámica (listas enlazadas, `Vector`...). En esta sección se considera la implementación con *arrays.* La declaración de una `Cola` ha de contener un *array* para almacenar los elementos de la cola y dos marcadores o apuntadores para mantener las posiciones `frente` y `fin` de la cola, es decir, un marcador apuntando a la posición de la cabeza de la cola y el otro al primer espacio vacío que sigue al final de la cola. Cuando un elemento se añade a la cola, se verifica si el marcador `fin` apunta a una posición válida, y entonces se añade el elemento a la cola y se incrementa el marcador `fin` en 1. Cuando un elemento se elimina de la cola, se hace una prueba para ver si la cola está vacía y, si no es así, se recupera el elemento de la posición apuntada por el marcador de cabeza, y éste se incrementa en 1.

La operación de poner un elemento en la cola comienza en la posición `fin` 0, y cada vez que se añade un nuevo elemento se incrementa `fin` en 1. La extracción de un elemento se hace por el extremo contrario, `frente`, y cada vez que se extrae un elemento avanza `frente` una posición. En la Figura 10.3 se puede observar cómo avanza el puntero `frente` al extraer un elemento.

El avance lineal de `frente` y `fin` tiene un grave problema, deja *huecos* por la *izquierda del array* . Llega a ocurrir que `fin` alcanza el índice mas alto del *array,* sin que puedan añadirse nuevos elementos, y, sin embargo, hay posiciones libres a la izquierda de `frente`.

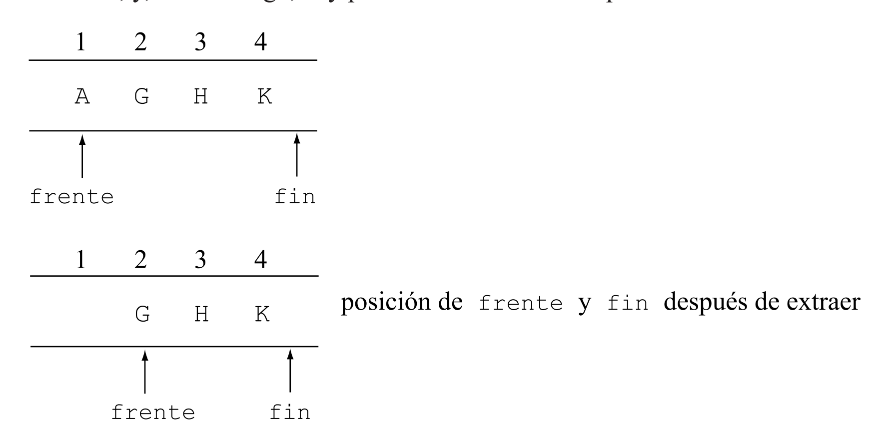


Figura 10.3 Una cola representada en un *array*

Una alternativa consiste en mantener fijo el `frente` de la cola al comienzo del *array* y mover todos los elementos de la cola una posición cada vez que se retira un elemento. Estos problemas quedan resueltos considerando el *array* como *circular*.

10.2.1. Declaración de la clase Cola

Una cola debe tratar elementos de diferentes tipos de datos: enteros, cadenas, objetos... Por esta circunstancia, los elementos han de ser de un tipo genérico. En Java todavía no está definido el tipo genérico; sin embargo, declarando el tipo del elemento `Object` y utilizando objetos y referencias se puede conseguir el mayor grado de genericidad.

La clase `ColaLineal` contiene un *array* `(listaCola)` cuyo máximo tamaño se determina por la constante `MAXTAMQ`. El tipo de los elementos queda sin especificar `(TipoDeDato)`, para que pueda sustituirse por un tipo simple, o bien por `Object`. Los atributos `frente` y `fin` son los punteros de cabecera y cola (fin), respectivamente. El constructor de la clase inicializa una cola vacía y define el *array:* `new` *TipoDeDato* `[MAXTAMQ]`.

La operación `insertar` toma un elemento y lo añade al final de la cola. `quitar` suprime y devuelve el elemento de la cabeza de la cola. La operación `frente` devuelve el elemento que está en la primera posición *(frente)* de la cola, sin eliminar el elemento.

La operación de control `colaVacia` comprueba si la cola tiene elementos, ya que es necesaria esta comprobación antes de eliminar un elemento. La operación `colaLlena` comprueba si la cola está llena, esta comprobación se realiza antes de insertar un nuevo miembro. Si las precondiciones para `insertar` y `quitar` se violan, el programa debe generar una excepción o error.

```
package TipoCola;

public class ColaLineal
{
   private static fin int MAXTAMQ = 39;
   protected int frente;
   protected int fin;
   protected TipoDeDato [] listaCola;

   public ColaLineal()
   {
      frente =   0;
      fin = -1;
      listaCola = new TipoDeDato [MAXTAMQ];
   }
      // operaciones de modificación de la cola
   public void insertar(TipoDeDato elemento) throws Exception
   {
      if (!colaLlena())
      {
         listaCola[++fin] = elemento;
      }
      else
         throw new Exception("Overflow en la cola");
   }
   public TipoDeDato quitar() throws Exception
   {
      if (!colaVacia())
      {
         return listaCola[frente++];
      }
```

```
         else
            throw new Exception("Cola vacia ");
      }
         //vacía la cola
      public void borrarCola()
      {
         frente =   0;
         fin = -1;
      }
         // acceso a la cola
      public TipoDeDato frenteCola() throws Exception
      {
         if (!colaVacia())
         {
            return listaCola[frente];
         }
         else
            throw new Exception("Cola vacia ");
      }
         // métodos de verificación del estado de la cola
      public boolean colaVacia()
      {
         return frente > fin;
      }
      public boolean colaLlena()
      {
         return fin == MAXTAMQ-1;
      }
   }
```

En la clase se hace referencia a un tipo no definido, *`TipoDeDato`*; es necesario sustituir ese identificador por el tipo verdadero de los elementos. Otra cuestión más importante es que esta implementación de una cola es notablemente ineficiente, ya que se puede alcanzar la condición de cola llena existiendo elementos del *array* sin ocupar. Esto se debe a que, al realizar la operación de `quitar` un elemento, avanza el `frente` y, por consiguiente, las posiciones anteriores quedan desocupadas, no accesibles. Una solución a este problema consiste en que al retirar el elemento, `frente` no se incremente y, a la vez, se desplacen el resto de elementos una posición a la izquierda.

Nota de programación

La realización de una cola con un *array* lineal es notablemente ineficiente, se puede alcanzar la condición de cola llena existiendo elementos del *array* sin ocupar. Se aconseja no utilizar esta implementación.

10.3. COLA CON UN *ARRAY* CIRCULAR

La alternativa, sugerida en la operación de `quitar` un elemento, de desplazar los restantes elementos del *array* de modo que la cabeza de la cola vuelva al principio del *array,* es costosa en términos de tiempo de computadora, especialmente si los datos almacenados en el *array* son estructuras de datos grandes.

La forma más eficiente de almacenar una cola en un *array* es modelarlo de tal forma que se una el extremo final con el extremo cabeza. Tal *array* se denomina *array circular* y permite que

la totalidad de sus posiciones se utilicen para almacenar elementos de la cola sin necesidad de desplazar elementos. La Figura 10.4 muestra un *array circular* de *n* elementos.

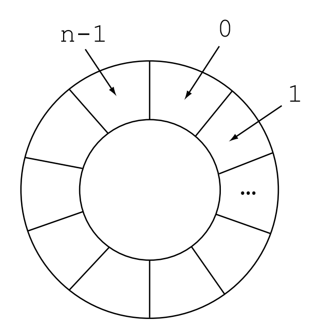


Figura 10.4 Un *array* circular

El *array* se almacena de modo natural en la memoria como un bloque lineal de *n* elementos. Se necesitan dos marcadores (apuntadores) *frente* y *fin* para indicar, respectivamente, la posición del elemento cabeza y del último elemento puesto en la cola.

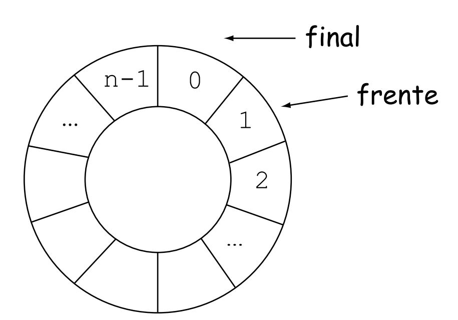


Figura 10.5 Una cola vacía

El `frente` siempre contiene la posición del primer elemento de la cola y avanza en el sentido de las agujas del reloj; `fin` contiene la posición donde se puso el último elemento y también avanza en el sentido del reloj (circularmente a la derecha). La implementación del movimiento circular se realiza según la *teoría de los restos,* de tal forma que se generen índices de `0` a `MAXTAMQ-1`:

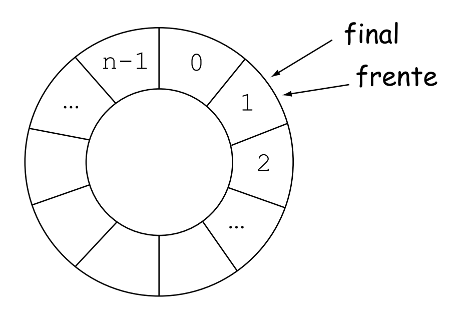


Figura 10.6 Una cola que contiene un elemento

Los algoritmos que formalizan la gestión de colas en un *array* circular han de incluir las operaciones básicas del *TAD* `Cola`, en concreto, las siguientes tareas básicas:

- Creación de una cola vacía, de tal forma que `fin` apunte a una posición inmediatamente anterior a `frente`:

  `frente = 0; fin  = MAXTAMQ-1.`

- Comprobar si una cola está vacía:

  `frente == siguiente(fin)`

- Comprobar si una cola está llena. Para diferenciar la condición de *cola llena* de *cola vacía* se sacrifica una posición del *array,* de tal forma que la capacidad de la cola va a ser `MAXTAMQ-1`. La condición de cola llena es:

  `frente == siguiente(siguiente(fin))`

- Poner un elemento a la cola: si la cola no está llena, avanzar `fin` a la siguiente posición, `fin = (fin + 1) % MAXTAMQ`, y asignar el elemento.
- Retirar un elemento de la cola: si la cola no está vacía, quitarlo de la posición `frente` y avanzar `frente` a la siguiente posición:`(frente + 1) % MAXTAMQ`.
- Obtener el elemento primero de la cola, si la cola no está vacía, sin suprimirlo de la cola.

10.3.1. Clase Cola con *array* circular

La clase declara los apuntadores `frente, fin` y el *array* `listaCola[]`. Para obtener la *siguiente* posición de una dada aplicando la *teoría de los restos*, se escribe el método `siguiente()`.

A continuación, se codifican los métodos que implementan las operaciones del *TAD Cola*. Ahora el tipo de los elementos es `Object`, de tal forma que se pueda guardar cualquier tipo de elementos.

```
package TipoCola;

public class ColaCircular
{
   private static fin int MAXTAMQ = 99;
   protected int frente;
   protected int fin;
   protected Object [] listaCola ;
      // avanza una posición
   private int siguiente(int r)
   {
      return (r+1) % MAXTAMQ;
   }
      //inicializa la cola vacía
   public ColaCircular()
   {
      frente = 0;
      fin = MAXTAMQ-1;
      listaCola  = new Object [MAXTAMQ];
   }
      // operaciones de modificación de la cola
   public void insertar(Object elemento) throws Exception
   {
      if (!colaLlena())
```

```
         {
            fin = siguiente(fin);
            listaCola[fin] = elemento;
         }
         else
            throw new Exception("Overflow en la cola");
      }
      public Object quitar() throws Exception
      {
         if (!colaVacia())
         {
            Object tm = listaCola[frente];
            frente = siguiente(frente);
            return tm;
         }
         else
            throw new Exception("Cola vacia ");
      }
      public void borrarCola()
      {
         frente = 0;
         fin = MAXTAMQ-1;
      }
         // acceso a la cola
      public Object frenteCola() throws Exception
      {
         if (!colaVacia())
         {
            return listaCola[frente];
         }
         else
            throw new Exception("Cola vacia ");
      }
         // métodos de verificación del estado de la cola
      public boolean colaVacia()
      {
         return frente == siguiente(fin);
      }
         // comprueba si está llena
      public boolean colaLlena()
      {
         return frente == siguiente(siguiente(fin));
      }
   }
```

Ejemplo 10.1

Encontrar un numero capicúa leído del dispositivo estándar de entrada.

El algoritmo para encontrar un número capicúa utiliza conjuntamente una *Cola* y una *Pila*. El número se lee del teclado, en forma de cadena con dígitos. La cadena se procesa carácter a carácter, es decir, dígito a dígito (un dígito es un carácter del 0 al 9). Cada dígito se pone en una cola y, a la vez, en una pila. Una vez que se terminan de leer los dígitos y de ponerlos en la *Cola* y en la *Pila,* comienza el paso de comprobación: se extraen en paralelo elementos de la cola y de la pila, y se comparan por igualdad. De producirse alguna no coincidencia entre dígitos, significa

que el número no es capicúa, y entonces se vacían las estructuras para pedir, a continuación, otra entrada. El número es capicúa si el proceso de comprobación termina con la coincidencia de todos los dígitos en orden inverso y tanto la pila como la cola quedan vacías.

¿Por qué utilizar una pila y una cola? Sencillamente porque aseguran que se procesan los dígitos en orden inverso; en la pila, el *último en entrar es el primero en salir;* en la cola, el *primero en entrar es el primero en salir.*

Se usa la clase `PilaVector` implementada con un `Vector` y la clase `ColaCircular` implementada con un *array circular.*

```
import TipoPila.PilaVector;
import TipoCola.ColaCircular;
import java.io.*;

public class Capicua
{

   public static void main(String [] a)
   {
      boolean capicua;
      BufferedReader entrada = new BufferedReader(
                          new InputStreamReader(System.in));
      String numero;

      PilaVector pila = new PilaVector();
      ColaCircular q = new ColaCircular();

      try {
         capicua = false;
         while (!capicua)
         {
            do {
               System.out.print("\nTeclea el número: ");
               numero = entrada.readLine();
            }while (! valido(numero)); // todos dígitos
            // pone en la cola y en la pila cada dígito
            for (int i = 0; i < numero.length(); i++)
            {
               Character c;
               c = new Character(numero.charAt(i));
               q.insertar(c);
               pila.insertar(c);
            }
            // se retira de la cola y la pila para comparar
            do {
               Character d;
               d = (Character) q.quitar();
               capicua = d.equals(pila.quitar());//compara por igualdad
            } while (capicua && !q.colaVacia());

            if (capicua)
               System.out.println(numero + " es capicúa. ");
            else
            {
               System.out.print(numero + " no es capicúa, ");
               System.out.println(" intente otro. ");
```

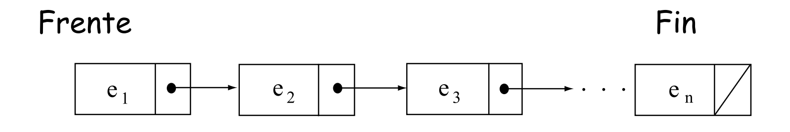
```
                // se vacía la cola y la pila
              q.borrarCola();
              pila.limpiarPila();
            }
          }
        }
        catch (Exception er)
        {
          System.err.println("Error (excepcion) en el proceso: " + er);
        }
      }
      private static boolean valido(String numero)
      {
        boolean sw = true;
        int i = 0;

        while (sw && (i < numero.length()))
        {
          char c;
          c = numero.charAt(i++);
          sw = (c >= '0' && c <= '9');
        }
        return sw;
      }
    }
```

10.4. COLA CON UNA LISTA ENLAZADA

La implementación del *TAD* `Cola` con un *array* necesita reservar memoria para el máximo de elementos previstos. En muchas ocasiones, esto da lugar a que se desaproveche memoria, pero también puede ocurrir los contrario, que se llene la cola y no se pueda seguir con la ejecución del programa a no ser que se amplíe la capacidad del *array.* Un diseño alternativo consiste en utilizar una lista enlazada que, en todo momento, tiene el mismo número de nodos que elementos la cola.

La implementación del *TAD Cola* con una lista enlazada permite ajustarse exactamente al número de elementos de la cola. Utiliza dos apuntadores (referencias) para acceder a la lista, `frente` y `fin`, que son los extremos por donde salen y por donde se ponen, respectivamente, los elementos de la cola.



e_1, e_2, ... e_n, son valores del tipo de los elementos

Figura 10.9 Cola con lista enlazada (representación gráfica típica)

La referencia `frente` apunta al primer elemento de la lista y, por tanto, de la cola (el primero en ser retirado). La referencia `fin` apunta al último elemento de la lista y también de la cola.

La lista enlazada crece y decrece según las necesidades, según se incorporen o se retiren elementos, entonces, en esta implementación no se considera el control de *Cola llena*.

10.4.1. Declaración de Nodo y Cola

La representación del *TAD* `Cola` con listas maneja dos clases; la clase `Nodo` y la clase, con las operaciones de las colas, `ColaLista`. El `Nodo` representa al elemento y al enlace con el siguiente nodo; al crear un `Nodo`, se asigna el elemento y el enlace se pone a `null`. Con el objetivo de generalizar, el elemento se declara de tipo `Object`.

La clase `ColaLista` define las variables (atributos) de acceso: `frente` y `fin`, y las operaciones básicas del *TAD* `Cola`. El constructor de `ColaLista` inicializa `frente` y `fin` a `null`, es decir, a la condición *cola vacía*.

```
package TipoCola;
   // declaración de Nodo (sólo visible en este paquete)
class Nodo
{
   Object elemento;
   Nodo siguiente;
   public Nodo(Object x)
   {
      elemento = x;
      siguiente = null;
   }
}
   // declaración de la clase ColaLista
public class ColaLista
{
   protected Nodo frente;
   protected Nodo fin;
   // constructor: crea cola vacía
   public ColaLista()
   {
      frente = fin = null;
   }
   // insertar: pone elemento por el final
   public void insertar(Object elemento)
   {
      Nodo a;
      a = new Nodo(elemento);
      if (colaVacia())
      {
         frente = a;
      }
      else
      {
         fin.siguiente = a;
      }
      fin = a;
   }
   // quitar: sale el elemento frente
   public Object quitar() throws Exception
   {
      Object aux;
      if (!colaVacia())
      {
         aux = frente.elemento;
         frente = frente.siguiente;
```

```
        }
        else
           throw new Exception("Eliminar de una cola vacía");
        return aux;
     }
     // libera todos los nodos de la cola
     public void borrarCola()
     {
        for (; frente != null;)
        {
           frente = frente.siguiente;
        }
        System.gc();
     }
     // acceso al primero de la cola
     public Object frenteCola() throws Exception
     {
        if (colaVacia())
        {
           throw new Exception("Error: cola vacía");
        }
        return (frente.elemento);
     }
     // verificación del estado de la cola
     public boolean colaVacia()
     {
        return (frente == null);
     }

  }
```

Ejercicio 10.1

Una variación del problema matemático llamado "problema de José" permite generar números de la suerte. Se parte de una lista de n *números; esta lista se va reduciendo siguiendo el siguiente algoritmo:*

1. *Se genera un número aleatorio* `n1`.
2. *Si* `n1 <= n` *se quitan de la lista los números que ocupan las posiciones* `1,1+n1,1+2*n1, ...;` `n` *toma el valor del número de elementos que quedan en la lista.*
3. *Se vuelve al paso 1.*
4. *Si* `n1 > n` *fin del algoritmo, los números de la suerte son los que quedan en la lista.*

El problema se va a resolver utilizando una *Cola*. En primer lugar, se genera una lista de `n` números aleatorios que se almacenan en la cola. A continuación, se siguen los pasos del algoritmo, en cada pasada se eliminan los elementos de la cola que están en las posiciones `(múltiplos de n1) + 1`. Estas posiciones `i` se pueden expresar matemáticamente de la siguiente forma:

```
i modulo n1 == 1
```

Una vez que termina el algoritmo, los números de la suerte son los que han quedado en la cola; entonces, se retiran de la cola y se escriben.

La *Cola* que se utiliza está implementada con listas enlazadas, y como el tipo de dato de los elementos es `Object`, los números enteros se ponen en la cola como referencias a objetos `Integer` que encierran el número entero.

```
import TipoCola.*;
import java.io.*;
import java.util.Random;

public class NumerosSuerte
{
   public static void main(String [] a)
   {
      int n, n1, n2, i;
      Integer nv;
      ColaLista q = new ColaLista();
      Random r = new Random();

      try
      {
            // número inicial de elementos de la lista
         n = 11 + r.nextInt(49);
            // se generan n números aleatorios
         for (i = 1; i <= n; i++)
         {
            nv = new Integer(1+r.nextInt(101));
            q.insertar(nv);
         }
            // se genera aleatoriamente el intervalo n1
         n1 = 1 + r.nextInt(11);
            // se retiran de la cola elementos a distancia n1
         System.out.print("\nSe quita de la lista: ");
         while (n1 <= n)
         {
            Object nt;
            n2 = 0; // contador de elementos que quedan
            for (i = 1; i <= n; i++)
            {
               nt = q.quitar();
               if (i % n1 == 1)
               {
                  System.out.print(nt + " ");
               }
               else
               {
                  q.insertar(nt); // se vuelve a meter en la cola
                  n2++;
               }
            }
            n = n2;
            n1 = 1 + r.nextInt(11);
         }
         System.out.println("\nLos números de la suerte: ");
         mostrarCola(q);
         System.out.println();
      }
```

```
         catch (Exception t)
         {
            System.out.println("Ha ocurrido la excepción: " + t);
         }
      }
         // extrae y escribe los elementos de la cola
      private static void mostrarCola(ColaLista q) throws Exception
      {
         while (!q.colaVacia())
         {
            Integer v;
            v = (Integer) q.quitar();
            System.out.print(" " + v.intValue());
         }
      }
   }
```

10.5. BICOLAS: COLAS DE DOBLE ENTRADA

La estructura *bicola* o *cola de doble entrada* se puede considerar que es una extensión del *TAD Cola*. Una bicola es un conjunto *ordenado* de elementos, al que se puede *añadir* o *quitar* elementos desde cualquier extremo del mismo. El acceso a la bicola está permitido desde cualquier extremo. Se puede afirmar que una bicola es una *cola bidireccional* .

Los dos extremos de una bicola se pueden identificar como `frente` y `fin` (iguales nombres que en una cola). Las operaciones básicas que definen una bicola son una ampliación de la operaciones que definen una cola:

CrearBicola	Inicializa una bicola sin elementos.
BicolaVacia	Devuelve `true` si la bicola no tiene elementos.
PonerFrente	Añade un elemento por el extremo frente.
PonerFinal	Añade un elemento por el extremo final.
QuitarFrente	Devuelve el elemento `Frente` y lo retira de la bicola.
QuitarFinal	Devuelve el elemento `Final` y lo retira de la bicola.
Frente	Devuelve el elemento que se encuentra en el extremo frente.
Final	Devuelve el elemento que se encuentra en el extremo final.

Al tipo de datos bicola se le puede poner restricciones respecto a la entrada o a la salida de elementos. Una *bicola con restricción de entrada* es aquella que sólo permite inserciones por uno de los dos extremos, pero que permite retirar elementos por los dos extremos. Una *bicola con restricción de salida* es aquella que permite inserciones por los dos extremos, pero sólo permite retirar elementos por uno de ellos.

La representación de una bicola puede ser con un *array,* con un *array circular,* con una *colección* tipo `Vector` o bien con *listas enlazadas*. Siempre se debe disponer de dos *marcadores* o variables índice *(apuntadores)* que se correspondan con ambos extremos, *frente* y *final,* respectivamente, de la estructura.

10.5.1. Bicola con listas enlazadas

La implementación del *TAD* `Bicola` con una lista enlazada se caracteriza por ajustar el tamaño al número de elementos; es una implementación dinámica, *crece* o *decrece* según lo requiera la ejecución del programa que utiliza la bicola. Para esta representación de la bicola es necesario declarar la clase `Nodo` con los atributos `elemento` y `siguiente` (enlace con el siguiente nodo), y la clase `Bicola` con los atributos `frente` y `fin` y los métodos que implementan las operaciones. Se observa que las operaciones del *TAD* `Bicola` son una extensión de las operaciones del *TAD* `Cola`, incluso son idénticas las variables de acceso: `frente` y `fin`. Por esta razón, resulta más eficiente implementar la clase `Bicola` con herencia de la clase `ColaLista`:

```
public class Bicola extends ColaLista
```

De esta forma, `Bicola` dispone de todos los métodos y atributos de la clase `ColaLista`. Entonces, sólo es necesario codificar las operaciones de `Bicola` que no están implementadas en `ColaLista`. Los métodos `ponerFinal()`, `quitarFrente()`, `bicolaVacia()`, `frenteBicola()` se corresponden con `insertar()`, `quitar()`, `colaVacia()` y `frenteCola()` de la clase `ColaLista`. Se añade el método `numElemsBicola()` para contar los elementos que están guardados. La implementación completa es la siguiente:

```
package TipoCola;

public class Bicola extends ColaLista
{
   // inicializa frente y fin a null
   public Bicola()
   {
      super();
   }
   // inserta por el final de la Bicola
   public void ponerFinal(Object elemento)
   {
      insertar(elemento); // método heredado de ColaLista
   }
   // inserta por el frente; método propio de Bicola
   public void ponerFrente(Object elemento)
   {
      Nodo a;
      a = new Nodo(elemento);
      if (colaVacia())
      {
         fin = a;
      }
      a.siguiente = frente;
      frente = a;
   }
   // retira elemento frente de la Bicola
   public Object quitarFrente() throws Exception
   {
      return quitar(); // método heredado de ColaLista
   }
   // retira elemento final; método propio de Bicola.
```

```
// Es necesario recorrer la bicola para situarse en
// el nodo anterior al final, para después enlazar.
public Object quitarFinal() throws Exception
{
   Object aux;
   if (!colaVacia())
   {
      if (frente == fin) // la Bicola dispone de un solo nodo
         aux = quitar();
      else
      {
         Nodo a = frente;
         while (a.siguiente != fin)
            a = a.siguiente;
         // siguiente del nodo anterior se pone a null
         a.siguiente = null;
         aux = fin.elemento;
         fin = a;
      }
   }
   else
      throw new Exception("Eliminar de una bicola vacía");
   return aux;
}
public Object frenteBicola()throws Exception
{
   return frenteCola();       // método heredado de ColaLista
}
// devuelve el elemento final
public Object finalBicola() throws Exception
{
   if (colaVacia())
   {
      throw new Exception("Error: cola vacía");
   }
   return (fin.elemento);
}
// comprueba el estado de la bicola
public boolean bicolaVacia()
{
   return colaVacia();      // método heredado de ColaLista
}
//elimina la bicola
public void borrarBicola()
{
   borrarCola();            // método heredado de ColaLista
}
public int numElemsBicola() // cuenta los elementos de la bicola
{
   int n;
   Nodo a = frente;
   if (bicolaVacia())
      n = 0;
   else
```

```
         {
            n = 1;
            while (a != fin)
            {
               n++;
               a = a.siguiente;
            }
         }
         return n;
      }
   }
```

Ejercicio 10.2

La salida a pista de las avionetas de un aeródromo está organizada en forma de fila (línea), con una capacidad máxima de 16 aparatos en espera. Las avionetas llegan por el extremo izquierdo (final) y salen por el extremo derecho (frente). Un piloto puede decidir retirarse de la fila por razones técnicas; en ese caso, todas las avionetas que la siguen han de ser quitadas de la fila, retirar el aparato y colocar de nuevo las avionetas desplazadas en el mismo orden relativo en el que estaban. La salida de una avioneta de la fila provoca que las demás sean movidas hacia adelante, de tal forma que los espacios libres del estacionamiento estén en la parte izquierda (final).

La aplicación para emular este estacionamiento tiene como entrada un carácter que indica una acción sobre la avioneta y la matrícula de la avioneta. La acción puede ser llegada (E), salida (S) de la avioneta que ocupa la primera posición y retirada (T) de una avioneta de la fila. En la llegada, si el estacionamiento está lleno la avioneta espera hasta que quede una plaza libre.

El estacionamiento se representa con una bicola (realmente, debería ser una bicola de salida restringida). ¿Por qué esta elección? La salida siempre se hace por el mismo extremo, sin embargo la entrada se puede hacer por los dos extremos, y así se simulan dos acciones: *la llegada* de una avioneta nueva y la *entrada de una avioneta que ha sido movida* para que salga una intermedia.

Las avionetas que se mueven para poder retirar del estacionamiento una que está en medio, se disponen en una *pila*. Así, la última en entrar será la primera en añadirse al extremo salida del estacionamiento (bicola) y seguir en el mismo orden relativo.

Las avionetas se representan mediante una cadena (`String`) con el número de matrícula. Entonces, los elementos de la pila y de la bicola son de tipo *cadena* (`String`).

La estructura pila que se utiliza es la implementada con una lista enlazada; está codificada en la clase `PilaLista`. Los elementos de la pila son de tipo `Object`, por consiguiente se ha de convertir `Object` a `String`. La bicola se representa mediante la clase `Bicola` del apartado anterior.

La clase con el método `main()` gestiona las operaciones indicadas en el enunciado del ejercicio. La resolución del problema no toma acción cuando una avioneta no puede incorporarse a la fila por estar llena. El lector puede añadir el código necesario para que, utilizando una cola, las avionetas que no pueden entrar en la fila se guarden en la cola y, cada vez que salga una avioneta, se añada otra desde la cola.

Clase principal

El método `retirar()` de la clase *principal* simula el hecho de que una avioneta, que se encuentra en cualquier posición de la bicola, decide salir de la fila; el método retira de la fila las avionetas por el *frente,* a la vez que las guarda en una pila (se guardan las avionetas que, temporalmente, tienen que *apartarse* para que salga la averiada) hasta que encuentra la avioneta a retirar. A continuación se insertan en la fila, por el *frente,* las avionetas de la pila, y así quedan en el mismo orden en el que estaban anteriormente. El atributo `maxAvtaFila (16)` guarda el número máximo de avionetas que pueden estar en la fila esperando la salida.

(El código fuente se encuentra en página web, archivo Ejercicio 10-2-web.)

RESUMEN

- Una cola es una lista lineal en la cual los datos se insertan por un extremo(*final*) y se extraen por el otro extremo *(frente)*. Es una estructura *FIFO (first-in, first-out*, primero en entrar-primero en salir).
- Las operaciones básicas que se aplican sobre colas son: `crearCola, colaVacia, colaLlena, insertar, frente, retirar.`
- `crearCola` inicializa a una cola sin elementos. Es la primera operación a realizar con una cola.
- `colaVacia` determina si una cola tiene o no elementos. Devuelve `true` si no tiene elementos.
- `colaLlena` determina si no se pueden almacenar más elementos en una cola. Se aplica esta operación cuando se utiliza un *array* para guardar los elementos de la cola.
- `insertar` añade un nuevo elemento a la cola, siempre por el extremo final.
- `frente` devuelve el elemento que está en el extremo frente de la cola sin extraerlo.
- `retirar` extrae el elemento frente de la cola.
- El *TAD* `Cola` se puede implementar con *arrays* y con listas enlazadas. La implementación con un *array* lineal es muy ineficiente; se ha de considerar el *array* como una estructura circular y aplicar la *teoría de los restos* para avanzar el frente y el final de la cola.
- La realización de una cola con listas enlazadas permite que el tamaño de la estructura se ajuste al número de elementos. La cola puede crecer indefinidamente, con el único tope de la memoria libre.
- Numerosos modelos de sistemas del mundo real son de tipo cola: la cola de impresión en un servidor de impresoras, los programas de simulación, las colas de prioridades en organización de viajes. Una cola es la estructura típica que se suele utilizar como almacenamiento de datos, cuando se envían datos desde un componente rápido de una computadora a un componente lento (por ejemplo, a una impresora).
- Las bicolas son colas dobles, las operaciones básicas de insertar y retirar elementos se pueden realizar por los dos extremos. A veces se ponen restricciones de entrada o de salida por algún extremo. Una bicola es, realmente, una extensión de una cola. La implementación natural del *TAD* `Bicola` es con una clase derivada de la clase `Cola`.

EJERCICIOS

10.1. Considerar una cola de nombres representada por un *array* circular con 6 posiciones, el campo frente con el valor: `frente = 2` y los elementos de la cola `Mar, Sella, Centurión`.

Escribir los elementos de la cola y los campos `frente` y `fin` según se realizan estas operaciones:

- Añadir `Gloria` y `Generosa` a la cola.
- Eliminar de la cola.
- Añadir `Positivo`.
- Añadir `Horche`.
- Eliminar todos los elementos de la cola.

10.2. A la clase que representa una cola implementada con un *array* circular y dos variables, `frente` y `fin`, se le añade una variable más que guarda el número de elementos de la cola. Escribir de nuevo los métodos de manejo de colas considerando este campo contador.

10.3. Una bicola es una estructura de datos lineal en la que la inserción y el borrado se pueden hacer tanto por el extremo `frente` como por el extremo `fin`. Suponer que se ha elegido una representación de los elementos con una lista doblemente enlazada y que los extremos de la lista se denominan `frente` y `fin`. Escribir la clase `Bicola` con la representación de los datos y la implementación de las operaciones del *TAD* `Bicola`.

10.4. Suponga que tiene ya codificados los métodos que implementan las operaciones del *TAD* `Cola`. Escribir un método para crear una copia de una cola determinada. Las operaciones que se han de utilizar serán únicamente las del *TAD* `Cola`.

10.5. Considere una bicola de caracteres, representada en un *array* circular. El *array* consta de 9 posiciones. Los extremos actuales y los elementos de la bicola son:

```
frente = 5   fin = 7          Bicola: A,C,E
```

Escribir los extremos y los elementos de la bicola según se realizan estas operaciones:

- Añadir los elementos `F` y `K` por el `final` de la bicola.
- Añadir los elementos `R`, `W` y `V` por el `frente` de la bicola.
- Añadir el elemento `M` por el `final` de la bicola.
- Eliminar dos caracteres por el `frente`.
- Añadir los elementos `K` y `L` por el `final` de la bicola.
- Añadir el elemento `S` por el `frente` de la bicola.

10.6. Se tiene una pila de enteros positivos. Con las operaciones básicas de pilas y colas escribir un fragmento de código para poner todos los elementos que son par de la pila en la cola.

10.7. Implementar el *TAD* `Cola` utilizando una lista enlazada circular. Por conveniencia, establecer el acceso a la lista, `lc`, por el último nodo (elemento) insertado y considerar al nodo siguiente de `lc` el primero o el que mas tarde se insertó.

PROBLEMAS

10.1. Escribir un programa en el que se generen 100 números aleatorios en el rango `–25 ... +25` y se guarden en una cola implementada mediante un *array* considerado circular. Una vez creada la cola, el usuario puede solicitar que se forme otra cola con los números negativos que tiene la cola original.

10.2. Escribir un método que tenga como argumentos dos colas del mismo tipo y devuelva cierto si las dos colas son idénticas.

10.3. Un pequeño supermercado dispone en la salida de tres cajas de pago. En el local hay 25 carritos de compra. Escribir un programa que simule el funcionamiento, siguiendo las siguientes reglas:

- Si cuando llega un cliente no hay ningún carrito disponible, espera a que lo haya.
- Ningún cliente se impacienta y abandona el supermercado sin pasar por alguna de las colas de las cajas.
- Cuando un cliente finaliza su compra, se coloca en la cola de la caja que hay menos gente, y no se cambia de cola.
- En el momento en que un cliente paga en la caja, su carrito de la compra queda disponible.

Representar la lista de carritos de la compra y las cajas de salida mediante colas.

10.4. En un archivo `F` están almacenados números enteros arbitrariamente grandes. La disposición es tal que hay un número entero por cada línea de `F`. Escribir un programa que muestre por pantalla la suma de todos los números enteros. Al resolver el problema habrá que tener en cuenta que, al ser enteros grandes, no pueden almacenarse en variables numéricas.

Utilizar dos pilas para guardar los dos primeros números enteros, almacenándose dígito a dígito. Al extraer los elementos de la pila, salen en orden inverso y, por tanto, de menor peso a mayor peso; se suman dígito con dígito y el resultado se guarda en una cola, también dígito a dígito. A partir de este primer paso se obtiene el siguiente número del archivo, se guarda en una pila y, a continuación, se suma dígito a dígito con el número que se encuentra en la cola; el resultado se guarda en otra cola. El proceso se repite, nuevo número del archivo se mete en la pila, que se suma con el número actual de la cola.

10.5. Una empresa de reparto de propaganda contrata a sus trabajadores por días. Cada repartidor puede trabajar varios días continuados o alternos. Los datos de los repartidores se almacenan en una lista enlazada. El programa a desarrollar contempla los siguientes puntos:

- Crear una cola que guarde el número de la seguridad social de cada repartidor y la entidad anunciada en la propaganda para un único día de trabajo.
- Actualizar la lista citada anteriormente (que ya existe con contenido) a partir de los datos de la cola.

La información de la lista es la siguiente: número de seguridad social, nombre y total de días trabajados. Además, está ordenada por el número de la seguridad social. Si el trabajador no está incluido en la lista, debe añadirse a la misma de tal manera que siga ordenada.

10.6. El supermercado *Esperanza* quiere simular los tiempos de atención al cliente a la hora de pasar por la caja. Los supuestos de los que se parte para la simulación son los siguientes:

- Los clientes forman una única fila. Si alguna caja está libre, el primer cliente de la fila es atendido. En el caso de que haya mas de un caja libre, la elección del número de caja por parte del cliente es aleatoria.
- El número de cajas del que se dispone para atención a los clientes es de tres, salvo que haya mas de 20 personas esperando en la fila; entonces se habilita una cuarta caja, que se cierra cuando no quedan clientes esperando. El tiempo de atención de cada una de las cajas está distribuido uniformemente: la caja 1 entre 1,5 y 2,5 minutos; la caja 2 entre 2 y 5 minutos, la caja 3 entre 2 y 4 minutos. La caja 4, cuando está abierta, tiene un tiempo de atención entre 2 y 4,5 minutos.
- Los clientes llegan a la salida en intervalos de tiempo distribuidos uniformemente, con un tiempo medio de 1 minuto.

El programa de simulación se debe realizar para 7 horas de trabajo. Se desea obtener una estadística con los siguientes datos:

- Clientes atendidos durante la simulación.
- Tamaño medio de la fila de clientes.
- Tamaño máximo de la fila de clientes.
- Tiempo máximo de espera de los clientes.
- Tiempo en que está abierto la cuarta caja.

CAPITULO 11

Colas de prioridades y montículos

Objetivos

Con el estudio de este capítulo, usted podrá:

- Conocer el tipo de datos *Cola de prioridades*.
- Describir las operaciones fundamentales de las colas de prioridades.
- Implementar colas de prioridades con listas enlazadas.
- Conocer la estructura de datos montículo binario.
- Implementar las operaciones básicas de los montículos con *complejidad logarítmica*.
- Aplicar un montículo para realizar una ordenación.

Contenido

Conceptos clave

- Árbol binario.
- Cola.
- Complejidad.
- Listas enlazadas.
- Montículo.
- Nodo *padre,* nodo *hijo*.
- Ordenación.
- Prioridad.

INTRODUCCIÓN

En este capítulo se estudian pormenorizadamente la estructuras de datos *colas de prioridades,* que se utilizan para planificar tareas en aplicaciones informáticas donde la prioridad de la tarea se corresponde con la clave de ordenación. También se aplican las *colas de prioridades* en los sistemas de simulación donde los sucesos se deben procesar en orden cronológico. El común denominador de estas aplicaciones es que siempre se selecciona el *elemento mínimo* una y otra vez hasta que no quedan mas elementos. Tradicionalmente, se ha designado la prioridad más alta con una clave menor, así los elementos de prioridad 1 tienen más prioridad que los elementos de prioridad 3.

Se utilizan estructuras lineales (listas enlazadas, vectores) para implementar las colas de prioridades; también, una estructura de multicolas. El montículo binario, o simplemente montículo, es la estructura más apropiada para implementar eficientemente las colas de prioridades.

La *idea intuitiva* de un montículo es la que mejor explica esta estructura de datos. En su *parte más alta* se encuentra el *elemento más pequeño;* los elementos descendientes de uno dado son mayores en la estructura montículo. La estructura montículo garantiza que el tiempo mínimo de las operaciones básicas insertar y eliminar, sea de complejidad logarítmica.

11.1. COLAS DE PRIORIDADES

El término cola sugiere la forma en que ciertos objetos esperan la utilización de un determinado servicio. Por otro lado, el término *prioridad* sugiere que el servicio no se proporciona únicamente aplicando el concepto *primero en llegar primero en ser atendido*, sino que cada objeto tiene asociada una prioridad basada en un criterio objetivo. Las colas de prioridades son una estructura ordenada que se utiliza para guardar elementos en un orden establecido. El orden para extraer un elemento de la estructura sigue estas reglas:

- Se elige la cola no vacía que se corresponde con la mayor prioridad.
- En la cola de mayor prioridad, los elementos se procesan según el orden de llegada: *primero en entrar-primero en salir*.

Una organización formada por colas de prioridades es el sistema de tiempo compartido, necesario para mantener una serie de procesos que esperan ser ejecutados por el procesador; cada proceso lleva asociado una prioridad. También las simulaciones de sucesos que ocurren en un tiempo discreto, como la atención de una fila de clientes en un sistema de *n* ventanillas de despacho de billetes de transporte, se realizan con una estructura de *colas de prioridades*.

11.1.1. Declaración del *TAD COLA DE PRIORIDAD*

La principal característica de una cola de prioridades consiste en que, repetidamente, se selecciona del conjunto de elementos el de clave máxima o prioridad máxima. Normalmente, se insertan nuevos elementos en la estructura a la vez que se procesan otros; esto lleva a especificar las operaciones que se detallan a continuación:

Tipo de dato *Proceso* que tiene asociado una *prioridad.*

Operaciones

`CrearColaPrioridad`	*Inicia la estructura con las prioridades sin elementos.*
`InserEnPrioridad`	*Añade un elemento a la cola según prioridad.*
`ElementoMin`	*Devuelve el elemento de la cola con la prioridad mas alta.*
`QuitarMin`	*Devuelve y retira el elemento de la cola con prioridad mas alta.*

`ColaVacía`	*Comprueba si una determinada cola no tiene elementos.*
`ColaPrioridadVacía`	*Comprueba si todas las colas de la estructura están vacías.*

A tener en cuenta

En programación siempre se ha seguido la convención de considerar un intervalo de máxima a mínima prioridad con valores clave en orden inverso; por ejemplo, asignar a la máxima prioridad la clave 0, a la siguiente la clave 1, y así sucesivamente de tal forma que la clave mínima sea n.

11.1.2. Implementación

Una forma simple de implementar una *cola de prioridades* es con una lista enlazada y ordenada según la prioridad. Otra alternativa consiste en un *array* o tabla de tantos elementos como prioridades estén previstas; cada elemento de la tabla guarda los objetos con la misma prioridad. Por último, se puede utilizar la estructura del montículo para guardar los componentes. La característica del montículo permite recuperar el elemento de máxima prioridad con una simple operación.

Los elementos de una *cola de prioridades* son objetos con la propiedad de *ordenación,* de tal forma que se puedan realizar comparaciones. Esto equivale a que dispongan de un atributo, de tipo ordinal, representando la prioridad del objeto.

11.2. TABLA DE PRIORIDADES

Quizás la forma más intuitiva de implementar una *cola de prioridades* es con una tabla o *array* de listas, donde cada elemento de la tabla se organiza a la manera de una cola *(primero en entrar-primero en salir)* y representa a los objetos con la misma prioridad. La Figura 11.1 muestra una tabla en la que cada elemento se corresponde con una prioridad y de la que emerge una cola.

La operación `inserEnPrioridad` añade un nuevo elemento `T` de prioridad `m` a la estructura, siguiendo estos pasos:

1. Buscar la prioridad `m` en la tabla.
2. Si existe, poner el elemento `T` al final de la cola `m`.
3. Si la prioridad `m` está fuera del rango de prioridades, generar un error

La operación `elementoMin` devuelve el elemento `frente` de la cola no vacía que tiene la máxima prioridad. La operación `quitarMin` también devuelve el elemento de máxima prioridad y, además, lo extrae de la cola.

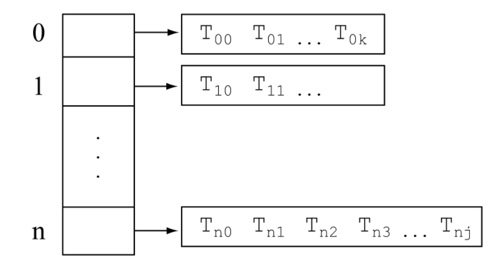


Figura 11.1 Cola de prioridades, de 0 a n prioridades

11.2.1. Implementación

La tabla se define como un *array* con el tamaño del número de prioridades. Se asume que estas prioridades varían en un rango de `0` al máximo previsto. Los elementos de la tabla son listas enlazadas con el comportamiento de una cola *(first-in, first-out)*. La cola contiene cualquier elemento de tipo `Object` y, como en ella se guardan elementos (objetos con prioridad) de la *cola de prioridades,* es necesario realizar conversión de tipo al recuperarlos.

Los elementos de la la *cola de prioridades* implementan la interfaz `Comparador`:

```
package ColaPrioridad;

public interface Comparador
{
   boolean igualQue(Object q);
   boolean menorQue(Object q);
   boolean menorIgualQue(Object q);
   boolean mayorQue(Object q);
   boolean mayorIgualQue(Object q);
}
```

Se declarara la clase `Tarea`, con los campos `item` y `prioridad`, para representar un objeto de la *cola de prioridades:*

```
package ColaPrioridad;

public class Tarea implements Comparador
{
   protected Object item;
   protected int prioridad;
   public Tarea(Object q, int n)
   {
      item = q;
      prioridad = n;
   }
   public int numPrioridad()
   {
      return prioridad;
   }
   public boolean igualQue(Object op2)
   {

      Tarea n2 = (Tarea) op2;
      return  prioridad == n2.prioridad;
   }
   public boolean menorQue(Object op2)
   {
      // orden inverso, es decir, prioridad 0 > prioridad 1
      Tarea n2 = (Tarea) op2;
      return  prioridad >  n2.prioridad;
   }
   public boolean menorIgualQue(Object op2)
   {
      Tarea n2 = (Tarea) op2;
      return  prioridad >= n2.prioridad;
   }
   public boolean mayorQue(Object op2)
```

```
    {
       Tarea n2 = (Tarea) op2;
       return  prioridad < n2.prioridad;
    }
    public boolean mayorIgualQue(Object op2)
    {
       Tarea n2 = (Tarea) op2;
       return  prioridad <= n2.prioridad;
    }

}
```

La clase que representa la tabla, a su vez, utiliza la clase `ColaLista,` del paquete `tipoCola` (apartado 10.4).

```
package ColaPrioridad;
import TipoCola. ColaLista;
public class ColaPrioridadT
{
   protected ColaLista []tabla;
   protected int maxPrioridad;
```

El constructor es el responsable de establecer el número de prioridades y definir el *array:*

```
public ColaPrioridadT(int n) throws Exception
{
   if (n < 1)
      throw new Exception ("Error en prioridad: " + n);
   maxPrioridad = n;
   tabla = new ColaLista [maxPrioridad + 1];
   for (int i = 0; i <= maxPrioridad; i++)
      tabla[i] = new ColaLista ();
}
```

Se supone que la máxima prioridad es `0` y la mínima, `maxPrioridad`. También, que hay una correspondencia biunívoca entre el índice de la tabla y el ordinal de la prioridad.

11.2.2. Insertar

La operación añade una nueva *tarea,* un elemento, a la *cola de prioridades*. La *tarea* se inserta en la cola `tabla[prioridad]`, siendo `prioridad` la asociada a la *tarea.*

```
public void inserEnPrioridad(Tarea t)throws Exception
{
   int p = t.numPrioridad();
   if (p >= 0 && p <= maxPrioridad)
   {
      tabla[p].insertar(t);
   }
   else
      throw new Exception("Tarea con prioridad fuera de rango");
}
```

La *complejidad* de añadir un elemento es la requerida por la operación de insertar en la cola `tabla[p]`. Por consiguiente, la operación tiene *complejidad constante* (tiempo constante).

11.2.3. Elemento de máxima prioridad

La operaciones `elementoMin` y `quitarMin` buscan, en primer lugar, el elemento de máxima prioridad; es una búsqueda de un elemento mínimo, ya que el convenio establecido es que la máxima prioridad se corresponde con `0`, y así sucesivamente. Una vez encontrado el índice de la cola con mayor prioridad, las operaciones `frente` y `quitar`, respectivamente, para `elementoMin` y `quitarMin` de la cola terminan el proceso.

```
public Tarea elementoMin()throws Exception
{
   int i = 0;
   int indiceCola = -1;
   // búsqueda de la primera cola no vacía
   do {
      if (!tabla[i].colaVacia())
      {
         indiceCola = i;
         i = maxPrioridad +1;  // termina el bucle
      }
      else
         i++;
   }while (i <= maxPrioridad);

   if (indiceCola != -1)
      return (Tarea) tabla[indiceCola].frenteCola();
   else
      throw new Exception("Cola de prioridades vacía");
}
```

La operación `quitarMin` sigue los mismos pasos que `elementoMin`, con la diferencia de que devuelve y retira el elemento `frente` de la cola de mayor prioridad:

```
return (Tarea) tabla[indiceCola].quitar();
```

La complejidad de las operaciones `frenteCola` y `quitar` de una cola es constante, y el proceso de búsqueda de la cola de máxima prioridad es lineal, por esa razón la complejidad de `elementoMin` y `quitarMin` es lineal.

11.2.4. Cola de prioridad vacía

La operación `colaPrioridadVacia` comprueba que cada una de las colas está vacía.

```
public boolean colaPrioridadVacia()
{
   int i = 0;
   while (tabla[i].colaVacia() && i < maxPrioridad)
      i++;
   return tabla[i].colaVacia();
}
```

11.3. VECTOR DE PRIORIDADES

La forma mas sencilla de implementar una *cola de prioridades* es mediante un `Vector` de objetos *(Tareas)* ordenado respecto a la prioridad del objeto. El vector se organiza de tal forma que un elemento *x* precede a *y* si:

1. *Prioridad(x)* es mayor que *Prioridad(y).*
2. Ambos tienen la misma prioridad, pero *x* se añadió antes que *y.*

Con esta organización, siempre el primer elemento del vector es el elemento de la cola de prioridades de máxima prioridad y el último elemento es el de menor prioridad y, por consiguiente, el último a procesar.

La clase que se declara para representar la *cola de prioridades* define el vector, y su constructor inicializa la estructura. Ahora, no es necesario establecer el número máximo de prioridades.

```
package ColaPrioridad;
import java.util.Vector;
public class ColaPrioridadV
{
   protected Vector cp;
   public ColaPrioridadV()
   {
      cp = new Vector();
   }
```

11.3.1. Insertar

La nueva *tarea* de la *cola de prioridades* se añade en la posición que le corresponda, teniendo en cuenta la prioridad de ésta. Es necesario recorrer el vector hasta encontrar la posición, `p`, de un elemento (*tarea*[1]) cuya prioridad sea menor. A continuación, se inserta en `p` la *tarea*.

```
public void inserEnPrioridad(Tarea t)throws Exception
{
   if (t.numPrioridad() < 0)
      throw new Exception("Tarea con prioridad fuera de rango");
    // búsqueda de la posición de inserción
   Tarea a;
   int p = 0;
   int n = cp.size();
   while (p < n)
   {
      a = (Tarea) cp.elementAt(p);
      if (a.numPrioridad() <= t.numPrioridad())
         p++;
      else
         n = p - 1;
   }
   cp.insertElementAt(t,p);// elementos posteriores son desplazados
}
```

En el peor de los casos, el *tiempo* de la búsqueda es lineal, al igual que el método `insertElementAt()`. Entonces, la operación es de *complejidad O(n) (complejidad lineal)*.

11.3.2. Elemento de máxima prioridad

Con esta representación de una cola de prioridades, el elemento de máxima prioridad siempre es el primero. La implementación de las operaciones `elementoMin` y `quitarMin` es directa.

```
public Tarea elementoMin()throws Exception
{
   if (colaPrioridadVacia())
```

1 Se ha preferido realizar las comparaciones directamente con la prioridad, pero se pueden utilizar las operaciones de la interfaz `Comparador`.

```
         throw new Exception("Cola de prioridades vacía");
      return (Tarea) cp.elementAt(0);
   }
   // elimina y devuelve el primer elemento
   public Tarea quitarMin()throws Exception
   {
      if (colaPrioridadVacia())
         throw new Exception("Cola de prioridades vacía");
      Tarea a = (Tarea)cp.elementAt(0);
      cp.removeElementAt(0);
      return a;
   }
```

La *complejidad* de la operación `elementoMin` es constante. La operación `quitarMin` es de *complejidad lineal* debido a que el método `removeElementAt()` desplaza, una posición a la izquierda, todos los elementos del vector.

11.3.3. Cola de prioridad vacía

Se comprueba la condición *cola prioridad vacía* con el tamaño del vector.

```
   public boolean colaPrioridadVacia()
   {
      return cp.size() == 0;
   }
```

11.4. MONTÍCULOS

El *montículo binario,* o simplemente *montículo,* es una estructura abstracta que almacena los datos de tal forma que pueden recuperarse en orden. Por ello, se utiliza para implementar las *colas de prioridades* en las que el orden de proceso es esencial, y también para uno de los algoritmos de ordenación más eficientes: *ordenación por montículos* o *HeapSort*. Además de la propiedad del orden, los montículos se caracterizan por la organización de los datos en un *array.*

La idea intuitiva de montículo es muy conocida: *una agrupación piramidal de elementos en la cual, para cualquier nivel, el peso de éstos es menor que el de los elementos adjuntos del nivel inferior y, por consiguiente, en la parte más alta se encuentra el elemento más pequeño.*

11.4.1. Definición de montículo

Un montículo binario *de tamaño n se define como un árbol binario*[2] *casi completo de n nodos, tal que el contenido de cada nodo es menor o igual que el contenido de su hijos*.

La definición formal de la estructura montículo está relacionada con los árboles binarios. Así, en la Figura 11.2 se puede observar un árbol binario con todos los niveles completos excepto el último, que puede no estar completo y tener los nodos situados lo más a la izquierda posible, que forma un montículo binario.

Sin embargo, el árbol binario de la Figura 11.3 no es completo, ya que en el último nivel no se han colocado los nodos de izquierda a derecha. Tampoco cumple la propiedad de ordenación de los montículos, ya que el nodo con clave `21` es mayor que uno de sus hijos, con clave `19`.

---

[2] El estudio de los árboles binarios se realiza en el Capítulo 13; se adelanta su concepto, por coherencia con la temática del capítulo.

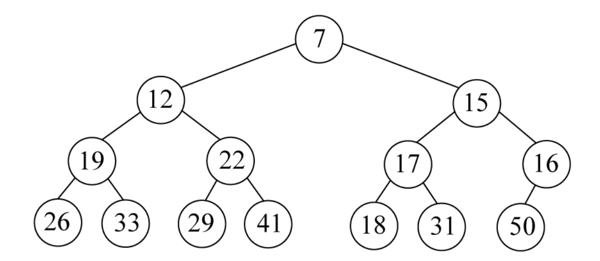


Figura 11.2 Montículo de elementos enteros

Definición

Un *árbol binario completo*[3] es un árbol con todos los niveles llenos con la excepción del último nivel, que se llena de izquierda a derecha. Tienen la propiedad importante de que la altura de un árbol binario completo de n nodos es $\lceil \log n \rceil$.

Los montículos tienen dos propiedades fundamentales: la propiedad de organizarse como un árbol binario y la propiedad de la ordenación. La primera asegura complejidades en las operaciones con cotas logarítmicas y, además, una fácil representación en una estructura tipo *array.*

La propiedad de ordenación conlleva que la clave de cualquier nodo sea inferior o igual a la de sus hijos (si tiene hijos). Por consiguiente, el nodo con menor clave debe ser el nodo raíz del árbol (siempre el nodo menor es el raíz) por ello la operación de *buscar mínimo* es directa, se puede implementar con *complejidad constante, O(1)*.

Las operaciones definidas con la estructura montículo pueden violar alguna de sus propiedades, si esto ocurre será necesario restablecer la condición de montículo.

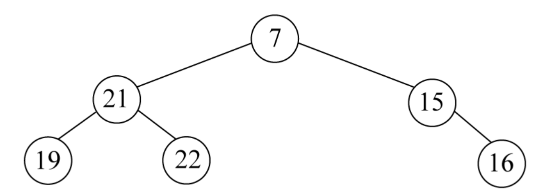


Figura 11.3 Árbol binario no completo

A tener en cuenta

En un montículo, la clave que se encuentra en la raíz es la menor de todos los elementos; sin embargo, no existe un orden entre las claves de los elementos que se encuentran en el mismo nivel.

11.4.2. Representación de un montículo

La característica de un árbol binario completo (tener los niveles internos llenos) permite guardar sus claves secuencialmente mediante un vector o *array.* La raíz del árbol se sitúa en la

[3] Véase la teoría completa de árboles binarios en el Capítulo 13.

posición[4] 0, sus hijos en las posiciones 1 y 2, los hijos de éstos en las posiciones 3, 4, 5 y 6, respectivamente, y así sucesivamente. El montículo de la Figura 11.4 consta de 8 nodos que han sido numerados según las posiciones que ocuparán en un almacenamiento secuencial.

Esta disposición natural de las claves de un árbol completo (las posiciones se corresponde con los niveles del árbol, para un mismo nivel de izquierda a derecha) en un *array* es muy útil, ya que permite acceder desde un nodo al nodo padre y a los hijos izquierdo y derecho (si los tiene). Para un nodo que esté en la posición `i`, su nodo padre ocupa la posición ⌈`i/2`⌋, el nodo hijo izquierdo se ubica en la posición `2*i+1` y el nodo hijo derecho en `(2*i+1)+1`. La representación secuencial del montículo binario de la Figura 11.4 sería:

```
15   39   18   41  72   37   19   56
```

A tener en cuenta

La forma secuencial de un montículo de `n` elementos implica que si `2*i+1 ≥ n` entonces `i` no tiene hijo izquierdo (tampoco hijo derecho), y si `(2*i+1)+1 ≥ n` entonces `i` no tiene hijo derecho.

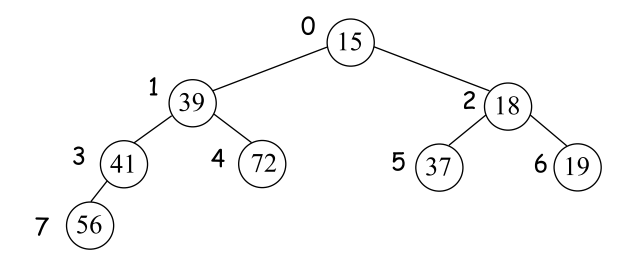


Figura 11.4 Montículo de 8 elementos numerando sus posiciones secuenciales

11.4.3. Propiedad de ordenación: Condición de montículo

La ordenación parcial de los nodos de un montículo se conoce como *condición de montículo*. Permite encontrar directamente el elemento mínimo, éste siempre se almacena en la primera posición del *array,* `v[0]`. Como consecuencia, el montículo es una estructura idónea para representar una *cola de prioridades*.

A tener en cuenta

La *condición de montículo* establece que cada nodo debe ser menor o igual que los nodos hijos. Esta condición deben satisfacerla todos los nodos del montículo, en definitiva, cada nodo será menor que sus descendientes.

---

[4] Se considera que el índice inferior de un *array* es 0; si fuera 1, habría que desplazar las posiciones.

Utilizando la representación secuencial de los montículos, la propiedad de ordenación parcial de las claves se expresa así:

```
v[i]   ≤   v[2*i+1]
                          ∀i = 0 .. n/2
v[i]   ≤   v[2*i+2]
```

El montículo de la Figura 11.2 se representa secuencialmente de la siguiente forma:

7	12	15	19	22	17	16	26	33	29	41	18	31	50

Cualquier nodo cumple la condición; así, `v[4] = 22` tiene como hijos a `v[9]` y `v[10]`, cuyos respectivos valores son `29` y `41`, menores que `22`.

11.4.4. Operaciones en un montículo

Se consideran las operaciones básicas `insertar`, `buscarMinimo`, `eliminarMinimo` (además de `crearMontículo` y `esVacio`); aquellas que modifican el montículo originan, posiblemente, una violación de la *condición de montículo*. Entonces, se necesitan operaciones auxiliares para recomponer el montículo; éstas recorren la estructura, de arriba abajo (de la raíz al último nivel), o bien en orden inverso.

Así, el montículo de la Figura 11.4 consta de `8` claves, y la inserción de una nueva clave, `32` por ejemplo, se realiza en la posición `8`, `v[8] = 32`:

15	39	18	41	72	37	19	56	**32**

La condición de ordenación es violada, ya que la clave de la posición `3` es mayor que la clave del hijo derecho, que acaba de ser insertado, `v[3] > v[8]`.

Los elementos de un montículo han de ser ordinales o tener la capacidad de establecer relaciones de comparación: *igual que*... Se declaran de tipo `Comparador`, de tal forma que las relaciones (<, ...) se hagan con las operaciones de la *interfaz;* entonces, los elementos de los montículos deberán implementar las operaciones de `Comparador`. El constructor de la clase establece una capacidad por defecto e inicializa los índices:

```
package monticuloBinario;
import ColaPrioridad.Comparador;
public class Monticulo
{
   static final int TAMINI = 20;
   private int numElem;
   private Comparador [] v;
   public Monticulo()
   {
      numElem = 0;
      v = new Comparador [TAMINI];
   }
```

11.4.5. Operación `insertar`

La operación añade una clave al montículo, incrementa su tamaño `(numElem)` y en el *hueco* se coloca la clave. De esta forma no se viola la estructura del montículo; ahora bien, también es

necesario comprobar la *condición de ordenación*. En el caso de que al situar la nueva clave no se viole la *condición de ordenación,* termina la operación. Por ejemplo, si de nuevo se considera el montículo[5] de la Figura 11.4 de `8` claves, la inserción de la clave `50` se realiza en la posición `8`, `v[8] = 50`, y termina la operación:

```
15  39  18  41  72  37  19  56  50
```

Para verificar la *condición de ordenación,* se ha de comprobar que la clave del nodo *padre,* sea menor o igual que la nueva clave insertada; en caso negativo, el *padre* (es decir, *padre* > *clave*) baja a la posición de la clave *(hueco)* y, naturalmente, ésta sube. Entonces, el proceso se repite, se vuelve a comparar la clave del nodo padre actual con la nueva clave, y si *padre* > *clave* se produce otro intercambio. En definitiva, la clave *sube por el árbol* hasta encontrar la posición que le corresponde. La búsqueda de la posición de inserción termina cuando la clave del *padre* es menor o igual que la nueva clave (no se viola la *condición de ordenación*) o bien se alcanza la raíz.

Al montículo binario de la Figura 11.5 se quiere añadir la clave `23`; en primer lugar, se hace un *hueco* (posición `8` del *array*) en el árbol.

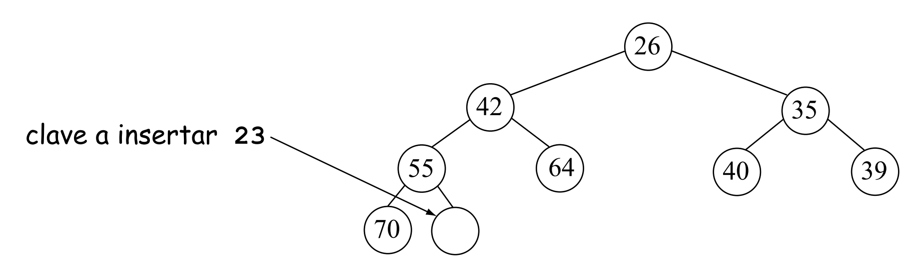


Figura 11.5 Montículo al que se añade la clave 23

El nodo padre, `55`, es mayor que `23`, por lo que viola la *condición de ordenación*. Baja la clave `55` al *hueco* y sube `23` al siguiente nivel del árbol, en busca de la posición de inserción. A continuación, se compara la clave `42` (nuevo nodo padre) con `23` y, como es mayor, baja `42` al *hueco* dejado anteriormente por `55`, y de nuevo sube `23` al siguiente nivel del árbol. El proceso continúa, se compara `26` con `23` y, al ser mayor, `26` baja al *hueco*. Se ha alcanzado la raíz del árbol, el proceso termina con `23` como nueva raíz del montículo. La Figura 11.6 muestra las comparaciones que se realizan hasta encontrar la posición donde insertar la clave `23`.

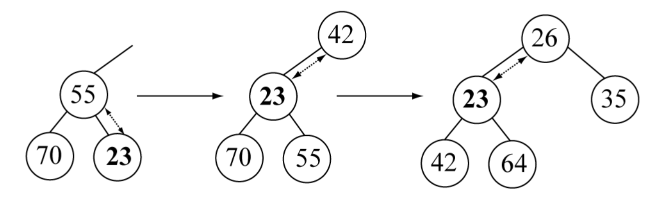


Figura 11.6 Reconstrucción del montículo moviendo hacia arriba la nueva clave

---

[5] Con el fin de facilitar la comprensión, los nodos de los montículos son valores enteros.

El número máximo de pasos que se realizan al insertar una clave en el montículo ocurre cuando la clave es el nuevo valor mínimo. Éste coincide con el número de niveles del árbol (para un montículo de n nodos, $\lceil \log\ n \rceil$); entonces el tiempo necesario, en el peor de los casos, para realizar la inserción es $O(\lceil \log\ n \rceil)$.

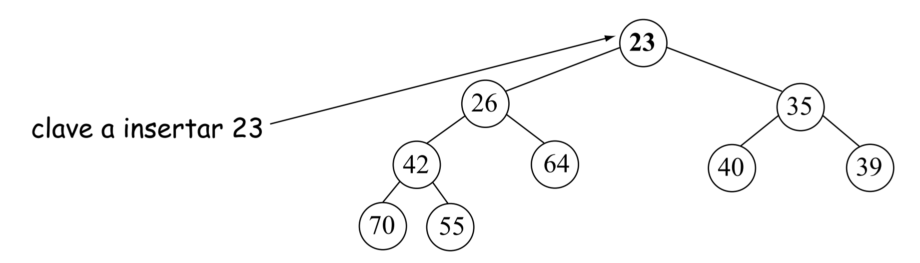


Figura 11.7 Montículo después de insertar 23

> ***A recordar***
>
> La inserción de una clave en el montículo se realiza en la siguiente posición libre del *array,* para a continuación hacer que *flote hacia arriba,* hasta encontrar la posición adecuada en el árbol. La complejidad de la operación es logarítmica, $O(\lceil \log\ n \rceil)$.

Implementación

Primero se codifican los métodos que devuelven las posiciones de nodo *padre*, *hijo izquierdo e hijo derecho:*

```
public static int padre(int i)
{
   return (i - 1)/ 2;
}
public static int hijoIzq(int i)
{
   return (2 * i + 1);
}
public static int hijoDer(int i)

   return (2 * i + 1) + 1;
}
```

El método `flotar()` realiza el *movimiento hacia arriba,* en busca de la posición de inserción. Tiene como argumento el índice, `i`, correspondiente al *hueco* donde, inicialmente, se encuentra la nueva clave. La comparación entre dicha clave y la del nodo padre, se realiza muy eficientemente, con acceso directo al *padre*. El bucle de búsqueda que realiza el proceso termina cuando la clave del nodo padre es menor o igual `(v[padre(i)] <= nuevaClave)` o bien se *ha subido* hasta la raíz `(i = 0)`.

```
private void flotar(int i)
{
   Comparador nuevaClave = v[i];
   while ((i > 0) && (v[padre(i)].mayorQue(nuevaClave)))
```

```
    {
       v[i] = v[padre(i)]; // baja el padre al hueco
       i = padre(i);       // sube un nivel en el árbol
    }
    v[i] = nuevaClave;     // sitúa la clave en su posición
}
```

El método `insertar()` comprueba que no esté lleno el montículo, asigna la clave en la primera posición libre *(hueco)*, llama a `flotar()` para que restablezca la *condición de ordenación* del montículo e incrementa el número de claves. Un montículo lleno no es un problema, ya que se amplía el vector en la cantidad `TAMINI`.

```
private boolean monticuloLleno()
{
   return (numElem == v.length);
}
private void ampliar()
{
   Comparador [] anteriorV = v;
   v = new Comparador [numElem + TAMINI];
   for (int j = 0; j < numElem; j++)
      v[j] = anteriorV[j];
}
public void insertar (Comparador clave)
{
   if (monticuloLleno())
   {
      ampliar();
   }
   v[numElem] = clave;
   flotar(numElem);
   numElem++;
}
```

11.4.6. Operación `buscar mínimo`

La propiedad de ordenación parcial de los montículos asegura que el elemento mínimo se encuentra en la raíz del árbol, que en el almacenamiento secuencial se corresponde con la primera posición del vector, `v[0]`.

```
public Comparador buscarMinimo() throws Exception
{
   if (esVacio())
      throw new Exception("Acceso a montículo vacío");
   return v[0];
}
```

La operación accede directamente al elemento, es de complejidad constante, *O(1)*.

11.4.7. Eliminar mínimo

La operación implica eliminar la clave que se encuentra en la raíz y, en consecuencia, reducir el número de elementos. Ahora bien, la estructura resultante tiene que seguir siendo un montículo, manteniendo la ordenación parcial de las claves y la forma estructural.

El elemento mínimo, `v[0]`, se extrae del *array,* quedando un *hueco* en esa posición. El número de elementos disminuye y, para que la estructura siga siendo un montículo, el último elemento (hoja del árbol más a la derecha) pasa a ser el elemento raíz. La Figura 11.8 muestra este primer paso en la eliminación del elemento mínimo.

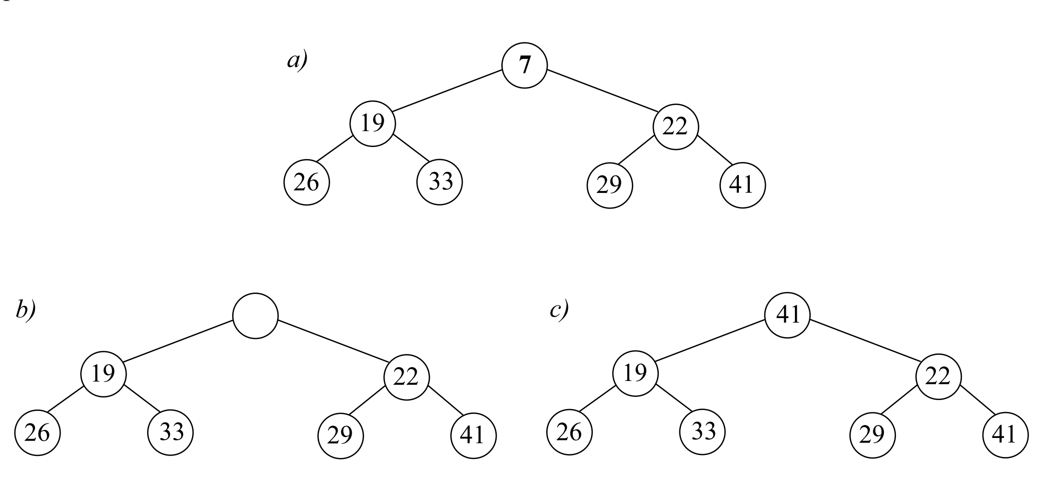


Figura 11.8 *a)* Montículo al que se elimina el mínimo `(7)`; *b) hueco* dejado, *c)* sube el último elemento `(41)`

Falta por analizar si la ordenación parcial, *condición de ordenación,* se mantiene. Posiblemente no será así, debido a que los últimos nodos son mayores que los del anterior nivel; por tanto, será necesario restablecer la *condición de ordenación*, para lo cual se procede a la inversa que en la inserción: se deja *hundir la clave por el camino de claves mínimas*.

El algoritmo compara el elemento raíz con el menor de sus hijos y, si la raíz es mayor, se intercambia con el menor de los hijos; así, en el montículo de la Figura 11.8*c,* la clave `41` se intercambia con `19`. Esto puede hacer que de nuevo se viole la *condición de ordenación* del montículo, pero en el siguiente nivel. Si esto ocurre, se procede de igual manera, se compara la clave con el menor de sus hijos y, si ésta es mayor, se intercambia con el menor. En el ejemplo de la Figura 11.9*b,* el menor de los hijos actuales de `41` es `26`, y se intercambian. Se observa que la clave que inicialmente subió a la raíz baja por *el camino de las claves mínimas;* el proceso continúa mientras no se viole la condición de montículo, o bien llegue al último nivel.

La Figura 11.9 muestra los intercambios que se realizan en el montículo de la Figura 11.8*c* hasta situar la clave `41` en la posición adecuada, en este caso como nodo hoja.

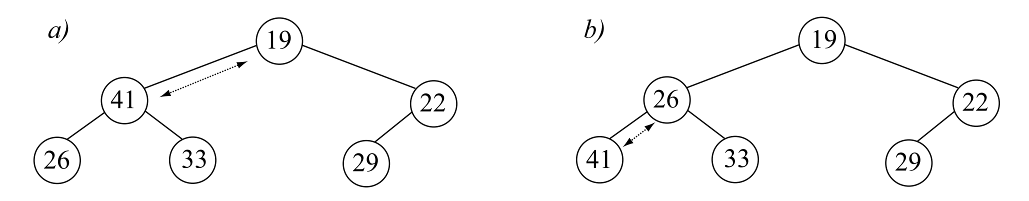


Figura 11.9 Proceso que restablece la *condición de ordenación* parcial del montículo. *a)* Intercambio de `41` con `19`; *b)* intercambio de `41` con `26`

A recordar

El procedimiento de hundir la clave situada en la raíz hasta encontrar la posición en la que no se viole la condición de montículo realiza un máximo de intercambios igual al número de niveles menos uno, por lo que la complejidad que tiene es *logarítmica*, $O(\lceil \log n \rfloor)$.

Implementación

El método `criba()` implementa el algoritmo, se pasa como argumento el índice del elemento `(raiz)` que se ha de dejar hundir.

```
public void criba (int raiz)
{
   boolean esMonticulo;
   int hijo;
   esMonticulo = false;
   while ((raiz < numElem / 2) && !esMonticulo)
   {
      // determina el índice del hijo menor
      if (hijoIzq(raiz) == (numElem - 1)) // único descendiente
         hijo = hijoIzq(raiz);
      else
      {
         if (v[hijoIzq(raiz)].menorQue(v[hijoDer(raiz)]))
            hijo = hijoIzq(raiz);
         else
            hijo = hijoDer(raiz);
      }
      // compara raiz con el menor de los hijos
      if (v[hijo].menorQue(v[raiz]))
      {
         Comparador t = v[raiz];
         v[raiz] = v[hijo];
         v[hijo] = t;
         raiz = hijo; /* continua por la rama de claves mínimas */
      }
      else
         esMonticulo = true;
   }
}
```

El método `eliminarMinimo()` realiza la operación: extrae la clave raíz y llama a `criba()` para restablecer la *condición de ordenación*.

```
public Comparador eliminarMinimo() throws Exception
{
   if (esVacio())
      throw new Exception("Acceso a montículo vacío");
   Comparador menor;
   menor = v[0];
   v[0] = v[numElem - 1];
   criba(0);
   numElem--;
   return menor;
}
```

Por último, se escribe el método `esVacio()` que verifica el estado del montículo.

```
public boolean esVacio()
{
   return numElem == 0;
}
```

11.5. ORDENACIÓN POR MONTÍCULOS *(HeapSort)*

Una de las aplicaciones que se ha dado a la estructura del montículo es la de ordenar los `n` elementos de un vector. Haciendo uso de las operaciones definidas anteriormente se puede diseñar un algoritmo de ordenación; los pasos a seguir son los siguientes:

1. Crear el montículo vacío `v[]`.
2. Insertar cada uno de los `n` elementos en el montículo. Se utiliza la operación `insertar`.
3. Extraer cada uno de los elementos, llamando a la operación `eliminarMinimo`, y asignarlos al *array* auxiliar `w[]`.

Un problema de este algoritmo es la necesidad de un segundo vector en el que almacenar los elementos que se extraen. Con el fin de realizar la ordenación en el mismo vector, sin necesidad de memoria adicional, se sigue una estrategia conocida como algoritmo de *ordenación HeapSort* o, simplemente, *ordenación por montículo*.

La operación `eliminarMinimo` del montículo retira el elemento menor y disminuye el número de elementos. El algoritmo *HeapSort* evita el uso de un segundo vector utilizando, en cada pasada, la actual última posición para guardar el elemento eliminado. La Figura 11.10*b* muestra el montículo después de eliminar la clave mínima y ser guardada en la última posición del *array;* la operación ha reconstruido el montículo, como consecuencia, el nuevo mínimo se corresponde con la clave `22`. En la Figura 11.10*c* se ha aplicado de nuevo la operación, el mínimo se guarda en la actual última posición, `4`, y la operación reconstruye el montículo. Con cuatro pasadas más, el *array* queda ordenado, aunque en orden decreciente.

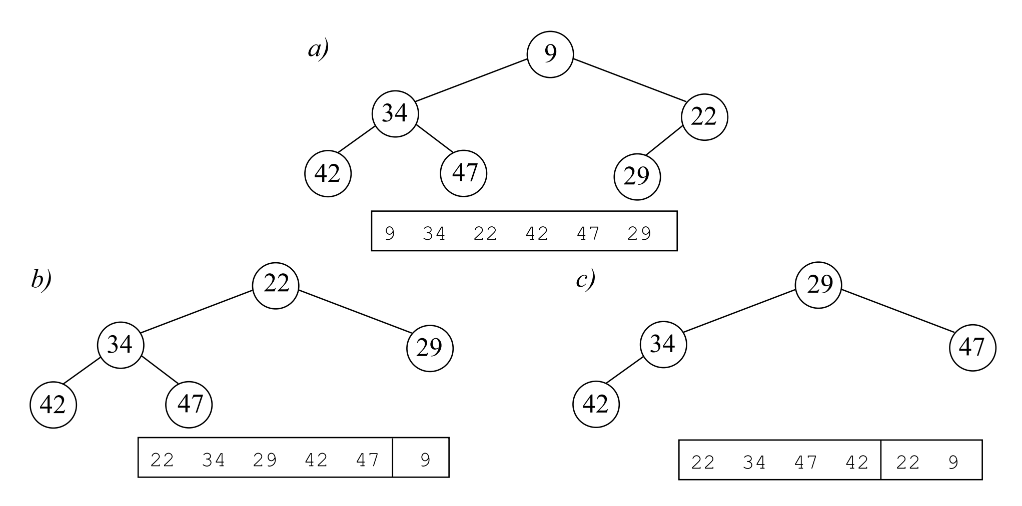


Figura 11.10 *a)* Montículo de 6 elementos; *b)* Montículo después de eliminar mínimo; *c)* Montículo después de volver aplicar eliminar mínimo

Los algoritmos de ordenación interna siempre ordenan en orden creciente. Para que el algoritmo del montículo lo haga así, simplemente hay que usar el *montículo maximal*. Éste cambia la *condición de ordenación,* de tal forma que la clave del nodo padre sea mayor o igual que la de sus hijos.

> **A tener en cuenta**
>
> El método de ordenación por montículos ordena en modo *descendente,* ya que siempre intercambia el elemento menor, `v[0]`, con el último elemento del montículo actual. Para una ordenación ascendente se considera *un montículo maximal,* simplemente se invierte el sentido de las desigualdades (cambiar < por >, y viceversa).

11.5.1. Algoritmo

El algoritmo empieza con un vector de `n` elementos que no cumple la condición de montículo. Lo primero que hace el algoritmo es construir el montículo de partida. Para ello, considera el montículo como una estructura recursiva. Se puede considerar que los nodos del montículo del último nivel del árbol son cada uno un *submontículo* de 1 nodo. Subiendo un nivel en el árbol, cada nodo es la raíz de un árbol que cumple la *condición del montículo,* excepto, quizás, en la raíz (su rama izquierda y derecha cumplen la condición, ya que se está *construyendo de abajo a arriba*); entonces, al aplicar el método `criba()` (reconstruye el montículo *hundiendo* la raíz), se asegura que cumple la *condición de ordenación* y ya es submontículo. El algoritmo va subiendo de nivel en nivel, construyendo tantos submontículos como nodos tiene el nivel, hasta llegar al primer nivel, en el que sólo hay un nodo, que es la raíz del montículo completo.

La Figura 11.11*a* se corresponde con el *array* inicial en la forma de árbol completo. En la Figura 11.11*b* todos los subárboles del penúltimo nivel son *montículos maximales,* se pueden observar los intercambios realizados para que cumplan la propiedad.

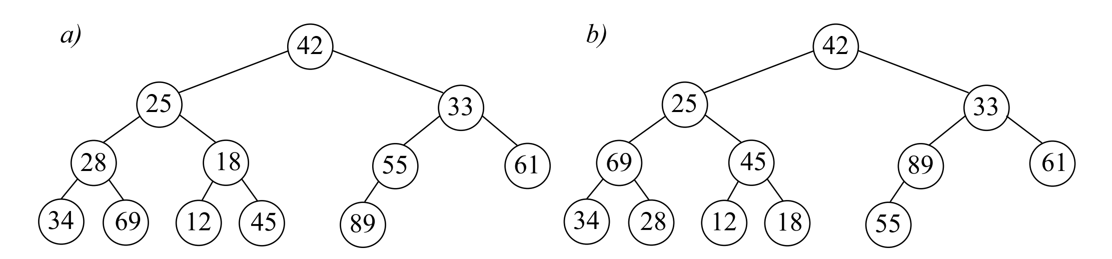


Figura 11.11 *a)* Montículo inicial; *b)* Montículo después de construir submontículos maximales en penúltimo nivel

En la Figura 11.12*a* se ha subido un nivel, se han construido dos submontículos maximales dejando hundir la correspondiente clave raíz (`25` y `33`, respectivamente). Para terminar, la Figura 11.12*b* muestra el montículo maximal completo; ésta última reconstrucción ha hundido la clave `42` y ha subido al elemento raíz la clave `89` que es el valor mayor.

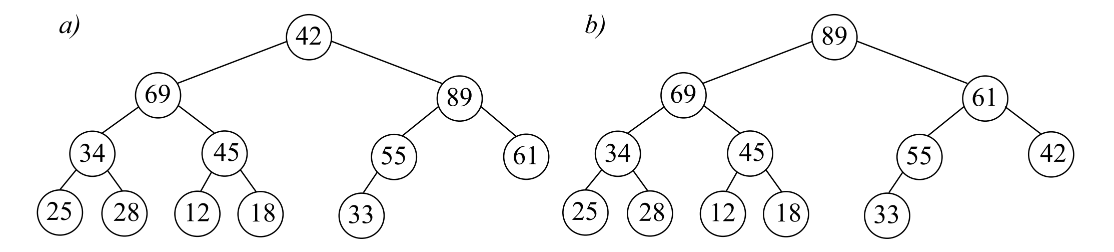


Figura 11.12 *a)* Reconstrucción del montículo en segundo nivel; *b)* montículo maximal reconstruido de abajo a arriba

Según esto, los pasos que sigue el algoritmo de ordenación por montículos o *HeapSort* para un *array* de `n` elementos `(0 .. n-1)` son:

1. Construir un montículo inicial con todos los elementos del vector:

   `v[0], v[1] , ..., v[n-1]`

2. Intercambiar los valores de `v[0]` y `v[n-1]`.
3. Reconstruir el montículo con los elementos `v[0], v[1], ... v[n-2]`.
4. Intercambiar los valores de `v[0]` y `v[n-2]`.
5. Reconstruir el montículo con los elementos `v[0], v[1], ... v[n-3]`.

Es un proceso iterativo que, partiendo de un montículo inicial, repite *intercambiar* los extremos, decrementar en `1` la posición del extremo superior y reconstruir el montículo del nuevo vector. En pseudocódigo:

```
OrdenacionMonticulo(v, n)

inicio
   <Construir monticulo inicial (v, 0, n)>
   desde k ← n-1 hasta 1 hacer
      intercambiar (v[0], v[k])
      reconstruir monticulo (v, 0, k - 1)
   fin _ desde
fin OrdenacionMonticulo
```

El problema de construir el montículo inicial y el de reconstruir se resuelve con el método `criba()`, que constituye la base del algoritmo.

11.5.2. Codificación

La clase `HeapSort` contiene los métodos `(static)` para realizar la ordenación. Se considera un *montículo maximal* para que, de esa forma, los elementos queden en orden ascendente. El método `criba()` cambia, trabaja con un *montículo maximal* y, además, tiene estos argumentos: *array* que se ordena, índice del elemento raíz e índice del último elemento. Los elementos del *array* deben implementar la interfaz `Comparador`.

Criba

```
public static void criba2 (Comparador v[], int raiz, int ultimo)
{
```

```
    boolean esMonticulo;
    int hijo;
    int numElem = ultimo + 1;
    esMonticulo = false;
    while ((raiz < numElem / 2) && !esMonticulo)
    {
       // determina el índice del hijo mayor
       if (Monticulo.hijoIzq(raiz) == (numElem - 1))
          hijo = Monticulo.hijoIzq(raiz);
       else
       {
          if (v[Monticulo.hijoIzq(raiz)].mayorQue(
                                            v[Monticulo.hijoDer(raiz)]))
             hijo = Monticulo.hijoIzq(raiz);
          else
             hijo = Monticulo.hijoDer(raiz);
       }
        //   compara raiz con el mayor de los hijos
       if (v[hijo].mayorQue(v[raiz]))
       {
          Comparador t = v[raiz];
          v[raiz] = v[hijo];
          v[hijo] = t;
          raiz = hijo; /* continua por la rama de claves máximas */
       }
       else
          esMonticulo = true;
    }
 }
```

Ordenación

Para construir el montículo inicial, se llama a `criba2()` pasando, sucesivamente, como segundo argumento, la raíz de los subárboles desde el penúltimo nivel del árbol hasta el raíz (posición `0`). En definitiva:

`criba2(v, j, n-1)` para todo `j = n/2, n/2 - 1, ..., 0`.

El bucle que construye el montículo es:

```
for (j = n / 2; j >= 0; j--)
   criba2(v, j, n-1);
```

En `criba2()` reside todo el trabajo de la realización del algoritmo de ordenación por montículos. Por último, la codificación del método de ordenación es:

```
public static void ordenacionMonticulo(Comparador v[], int n)
{
   int j;
   for (j = n / 2; j >= 0; j--)
      criba2(v, j, n - 1);

   for (j = n - 1; j >= 1; j--)
   {
      Comparador t;
```

```
        t = v[0];
        v[0] = v[j];
        v[j] = t;
        criba2(v, 0, j-1);
    }
}
```

11.5.3. Análisis del algoritmo de ordenación por montículos

El tiempo del algoritmo de ordenación *HeapSort* depende de la complejidad del método `criba2()`. Para determinar ésta, supóngase que el árbol binario que representa al montículo está completo, y que el número de niveles de que consta es `k`. Además, el nivel que se corresponde con el elemento raíz es el cero, por lo que los niveles de que consta varían de `0` a `k-1 (0, 1, 2, 3, ... k-1)`.

El número de elementos de cada nivel es potencia de `2`: nivel `0`, $2^0 = 1$; nivel `1`, $2^1 = 2$, $2^2 = 4$, $2^3 = 8$, ... 2^{k-1}. Entonces, el total de elementos es:

$$1 + 2 + 2^2 + 2^3 + ... + 2^{k-1}$$

Si *n* es el número de elementos a ordenar, aplicando la fórmula de la suma de los términos de una progresión geométrica de razón `2` se ha de cumplir:

$$n = \frac{1 * 2^{k-1}}{2}$$

y tomando logaritmos se obtiene $k = log_2 n + 1$.

`criba2()` realiza, en el peor de los casos, tantas iteraciones como niveles tiene el árbol menos `1`, hasta que llega al penúltimo nivel *($log_2 n$)*. Por ello, se puede concluir que la complejidad es *O(log n), complejidad logarítmica*.

La ordenación consta de dos bucles, el primero:

```
for (j = n / 2; j >= 0; j--)
```

construye el montículo inicial y se ejecuta *n/2* veces, por esa razón, su complejidad:

$O(n/2 * log\ n)$

El segundo bucle, `for (j = n - 1; j >= 1; j--)`, se ejecuta *n-1* veces y, por consiguiente, su complejidad es:

$O((n-1) * log\ n)$

Se puede concluir que la eficiencia del método de ordenación es:

$O(n/2 * log\ n) + O((n-1) * log\ n)$

Por las propiedades de la notación *O,* se puede afirmar que la complejidad del método de ordenación *HeapSort* es:

$O(n * log\ n)$

Este método de ordenación es de los más eficientes, tiene la misma complejidad que el método *Quicksort*.

11.6. COLA DE PRIORIDADES EN UN MONTÍCULO

La estructura de montículo es la más eficiente para implementar una cola de prioridades. A continuación se escribe su implementación utilizando las operaciones básicas del montículo. Naturalmente, los elementos son referencias a `Tarea` que implementa la interfaz `Comparador`.

```
package ColaPrioridad;
import monticuloBinario.Monticulo;
public class ColaPrioridadM
{
   protected Monticulo cp;
   public ColaPrioridadM()
   {
      cp = new Monticulo();
   }
   // añade una tarea a la cola
   public void inserEnPrioridad(Tarea t)throws Exception
   {
      cp.insertar(t);
   }
   public Tarea elementoMin ()throws Exception
   {
      return (Tarea)cp.buscarMinimo();
   }
   public Tarea quitarMin()throws Exception
   {
      return (Tarea)cp.eliminarMinimo();
   }
   public boolean colaPrioridadVacia()
   {
      return cp.esVacio();
   }

}
```

11.6.1. Ejemplo de *cola de prioridades*

Se desarrolla un supuesto cuyo único fin es escribir una aplicación de la estructura *cola de prioridades* con un montículo. Considera eventos representados por una cadena, que lo describe, y el tiempo de espera (entero que toma valores a partir de `0`). La planificación de eventos depende del tiempo, a menor tiempo más prioridad. La clase `Evento` describe las características de un evento, es una extensión de la clase `Tarea` (véase el apartado 11.2.1).

```
import ColaPrioridad.*;
import java.io.*;

class Evento extends Tarea
{
   public Evento (String proces, int tiempo)
   {
      super(proces,tiempo);
   }
      /*
         prioridad orden inverso de tiempo; es decir, tiempo 0
         mayor prioridad que tiempo 1.
```

```
      */
   public String toString()
   {
      String proces;
      int tiempo;
      proces = (String)super.item;
      tiempo = super.prioridad;
      return  proces + " tiempo: " + tiempo;
   }
}
```

La aplicación crea eventos secuencialmente, los inserta en la cola de prioridades. A continuación, se procesan (se muestran) en orden.

```
import ColaPrioridad.ColaPrioridadM;
import java.io.*;
import java.util.Random;

public class SimulaColaPrioridad
{
   public static void main(String [] ar)
   {
      String pt;
      int tm;
      BufferedReader entrada = new BufferedReader(
                        new InputStreamReader(System.in));
      Random a = new Random();
      ColaPrioridadM quePrio = new ColaPrioridadM();

      try {
         // bucle para crear eventos e insertarlos
         int cuenta = 0;
         Evento evento;
         do {
            System.out.print("Descripción: ");
            pt = entrada.readLine();
            tm = a.nextInt(15);   //simula tiempo de proceso
            evento = new Evento(pt,tm);
            quePrio.inserEnPrioridad(evento);
            cuenta++;
         } while (cuenta < 15);
         // listado de eventos en orden
         System.out.println("Procesos en orden de prioridad");
         System.out.println("-----------------------------");
         while (!quePrio.colaPrioridadVacia())
         {
            evento = (Evento)quePrio.quitarMin();
            System.out.println(evento);
         }
      }
      catch (Exception er)
      {
         System.err.println("Error de acceso a cola: " + er);
      }
   }
}
```

RESUMEN

Las *colas de prioridades* son estructuras de datos que tienen la propiedad de que siempre se accede al *mínimo elemento*. Una cola de prioridad puede asemejarse a una estructura de *n* colas, tantas como prioridades se quieran establecer en la estructura. La prioridad asignada a cada elemento es la clave de ordenación parcial de los elemento que forman parte de una *cola de prioridades*. Las operaciones básicas que tiene esta estructura son: *InserEnPrioridad, elemento mínimo, quitar mínimo*. La primera añade un elemento a la estructura en el orden que le corresponde según la prioridad y a igual prioridad el último. Las otras dos operaciones acceden al elemento mínimo, en cuanto a la clave, que en realidad es el que tiene mayor prioridad.

En el capítulo se han representado las colas de prioridades con una estructura multienlazada formada por un *array* de tantas colas como prioridades. A su vez, cada cola implementada con una lista enlazada. También con un `Vector` ordenado respecto a la prioridad de cada elemento.

Tradicionalmente, las colas de prioridades se han implementado con una estructura de árbol binario, llamada *montículo binario*. La definición que se ha dado a esta estructura se corresponde con los montículos minimales, los cuales tienen en la raíz el elemento mínimo; también existe la estructura de montículo maximal, que en vez de tener el mínimo en la parte más alta tienen el máximo. Las operaciones definidas sobre los montículos: *insertar*, *buscar mínimo* y *eliminar mínimo* se han implementado representado el montículo en un *array* (índice `0` primera posición), de tal forma que siendo `i` el índice de un nodo, `2*i+1` y `(2*i+1)+1` son los índices del nodo hijo izquierdo y derecho respectivamente.

Como aplicación del montículo maximal, el capítulo desarrolla el algoritmo de ordenación *HeapSort,* también llamado *ordenación por montículos*. Se analiza la complejidad del algoritmo, siendo esta `O(n*log)`, por tanto, está entre los algoritmos de ordenación más eficientes y además es fácil de implementar.

EJERCICIOS

11.1. El *TAD Cola de prioridad* se ha implementado utilizando una tabla de colas, tantas como prioridades. Escribir las operaciones `inserEnPrioridad` y `quitarMin` considerando que se utiliza una lista enlazada en la que se guardan, en orden de prioridad, los elementos de la cola de prioridad.

11.2. Describa un algoritmo para implementar la operación `cambiar()` que sustituya la clave del elemento `k` en un montículo.

11.3. Demuestre la certeza de la siguiente afirmación: en un árbol binario completo de *n* claves, hay exactamente *n/2* hojas.

11.4. Dibujar un montículo binario maximal a partir de un montículo vacío, al realizar las siguientes operaciones: `insertar(10)`, `insertar(50)`, `insertar (20)`, `insertar(60)`, `eliminarMinimo()`, `insertar(70)`, y, por último, `insertar(30)`.

11.5. Considere el árbol completo de la Figura 11.13; suponiendo que se guarda secuencialmente en el *array* `v[]`, encontrar el montículo que se forma llamando `insertar()` de la estructura montículo, transmitiendo en cada llamada la clave `v[k]` para valores `k = 0, 1, ... n-1`; n es el número de nodos.

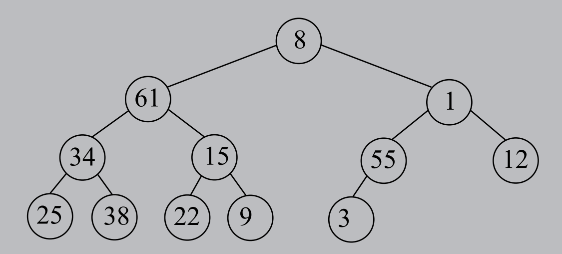


Figura 11.13 Árbol binario completo de 12 nodos

11.6. Mostrar el resultado del algoritmo de ordenación por montículos sobre la entrada: `18, 11, 22, 33, 11, 34, 44, 2, 8, 11.`

11.7. Diseñar un algoritmo para que, dados dos montículos, binarios se mezclen formando un único montículo. ¿Qué complejidad tiene el algoritmo diseñado?

11.8. Suponer que se quiere añadir la operación `eliminar(k)`, con el objetivo de quitar del montículo el elemento que se encuentra en la posición `k`. Diseñar un algoritmo que realice la operación.

11.9. En un montículo minimal, diseñar un algoritmo que encuentre el elemento con mayor clave. ¿Qué complejidad tiene el algoritmo diseñado?

PROBLEMAS

11.1. Escribir un programa que compare la eficiencia de los métodos de ordenación *HeapSort* y *Quicksort* para:

a) 1.000 elementos ordenados en modo ascendente.

b) 1.000 elementos ordenados en modo descendente.

c) 1.000 elementos aleatorios.

11.2. Implementar el algoritmo escrito en el Ejercicio 11.2 para la operación *cambiar* un elemento que ocupa la posición `k`.

11.3. La Universidad de La Alcarria dispone de 15 computadores conectados a Internet. Se quiere hacer una simulación de la utilización de los computadores por los alumnos. Para ello se supone que la frecuencia de llegada de un alumno es de 18 minutos las dos primeras horas, y de 15 minutos el resto del día. El tiempo de utilización del computador es un valor aleatorio, entre 30 y 55 minutos. El programa debe tener como salida líneas en las que se refleje la llegada de un alumno, la hora en que llega y el tiempo de la conexión. En el supuesto de que llegue un alumno y no haya computadores libres, el alumno

no espera, se mostrará el correspondiente aviso. En una cola de prioridad se tiene que guardar los distintos “eventos” que se producen, de tal forma que el programa avance de evento a evento. Suponer que la duración de la simulación es desde las 10 de la mañana a las 8 de la tarde.

11.4. La entrada a una sala de arte que ha inaugurado una gran exposición sobre la evolución del arte rural se realiza por tres torniquetes. Las personas que quieren ver la exposición forman una única fila y llegan de acuerdo a una distribución exponencial, con un tiempo medio entre llegadas de 2 minutos. Una persona que llega a la fila y ve esperando a más de 10 personas, se va con una probabilidad del 20%, aumentando esta en 10 puntos por cada 15 personas mas que haya esperando, hasta un tope del 50%. El tiempo medio que tarda una persona en pasar es de 1 minuto (compra de la entrada y revisión de los bolsos). Además, cada visitante emplea en recorrer la exposición entre 15 y 25 minutos distribuidos uniformemente.

La sala sólo admite, como máximo, 50 personas. Simular el sistema durante un periodo de 6 hora para determinar:

- Número de personas que llegan a la sala y número de personas que entran.
- Tiempo medio que debe esperar una persona para entrar en la sala.

Utilizar para simulación el *TAD Cola de prioridad*.

CAPITULO 12

Tablas de dispersión, funciones *hash*

Objetivos:

Con el estudio de este capítulo, usted podrá:

- Distinguir entre una tabla lineal y una tabla *hash.*
- Conocer las aplicaciones que tienen las tablas *hash.*
- Manejar funciones matemáticas sencillas con las que obtener direcciones según una clave.
- Conocer diversos métodos de resolución de colisiones.
- Realizar en Java una tabla *hash* con resolución de colisiones.

Contenido

Conceptos clave

- ♦ Acceso aleatorio.
- ♦ Colisión.
- ♦ Diccionario.
- ♦ Dispersión.
- ♦ Exploración.
- ♦ Factor de carga.
- ♦ Hueco.
- ♦ Tabla.

INTRODUCCIÓN

Las tablas de datos permiten el acceso directo a un elemento de una secuencia indicando la posición que ocupa. Un *diccionario* es también una secuencia de elementos, pero éstos son, realmente, pares formados, por ejemplo, por un identificador y un número entero. Las tablas de dispersión se suelen utilizar para implementar cualquier tipo de *diccionario,* por ejemplo, la tabla de símbolos de un compilador necesaria para generar el código ejecutable. La potencia de las tablas *hash* o dispersas radica en la búsqueda de elementos; conociendo el campo clave se puede obtener directamente la posición que ocupa y, por consiguiente, la información asociada a dicha clave. Sin embargo, no permiten algoritmos eficientes para acceder a todos los elementos de la tabla en su recorrido. El estudio de tablas *hash* acarrea el estudio de funciones *hash* o de dispersión, que mediante expresiones matemáticas permiten obtener direcciones según una clave, que es el argumento de la función.

En este capítulo se estudian diversas funciones *hash* y cómo resolver el problema de obtener una misma dirección con dos o más claves, lo que se conoce como colisión.

12.1. TABLAS DE DISPERSIÓN

Las *tablas de dispersión* o, simplemente *tablas hash,* son estructuras de datos que se usan en aplicaciones que manejan una secuencia de elementos, de tal forma que cada elemento tiene asociado un valor *clave,* que es un número entero positivo perteneciente a un rango de valores, relativamente pequeño. En estas organizaciones, cada uno de los elementos ha de tener una clave que identifica de manera unívoca al elemento. Por ejemplo, el campo *número de matricula* del conjunto de alumnos puede considerarse un campo clave para organizar la información relativa al alumnado de una universidad. El número de matrícula es único, hay una relación biunívoca, uno a uno, entre el campo y el registro alumno. Puede suponerse que no existen, simultáneamente, dos registros con el mismo número de matrícula.

Definición

Un *Diccionario* es un tipo abstracto de datos en el que los elementos tienen asociada una clave única en el conjunto de los números enteros positivos, de tal forma que, para cualquier par de elementos distintos, sus claves son también distintas. Con las tablas de dispersión se implementa eficientemente el tipo abstracto de datos *Diccionario*.

12.1.1. Definición de una tablas de dispersión

Las tablas de dispersión son estructuras de datos que tienen como finalidad realizar las operaciones fundamentales de búsqueda y eliminación de un registro en un tiempo de ejecución constante (complejidad constante `O(1)`). La organización ideal de una tabla es de tal forma que el campo clave de los elementos se corresponda directamente con el índice de la tabla. Por ejemplo, una compañía tiene `300` empleados, cada uno identificado con un número de nómina de `0` a `999`. La forma de organizar la tabla es con un *array* de `1000` registros:

```
Empleado[] tabla = new Empleado[1000];
```

El elemento `tabla[i]` almacena al empleado cuya nómina es `i`. Con esta organización, la búsqueda de un empleado es directa, con un único acceso, debido a que el número de nómina es la posición en la tabla. La eficiencia se puede expresar como tiempo constante, `O(1)`.

Sin embargo, muchas posiciones de la tabla están vacías, se corresponden con números de nómina que no existen. Y eso que el rango del campo clave es relativamente pequeño; si los números de nómina fueran de 5 dígitos, las posiciones vacías estarían en clara desproporción, y la memoria ocupada por la tabla queda desaprovechada. Pero enseguida se puede plantear una solución: tomar los tres primeros dígitos del número de la nómina, campo clave, como índice del *array* o tabla de registros, de este modo se ha hecho una transformación del campo clave en un entero de 3 dígitos:

```
h(número de nómina) → indice
```

Se puede concluir que el primer problema que plantea esta organización: ¿cómo evitar que el *array* o vector utilizado esté en una proporción adecuado al número de registros? Las funciones de transformación de claves, funciones *hash,* permiten que el rango posible de índices esté en proporción al número real de registros.

Las funciones que transforman números grandes en otros más pequeños se conocen como *funciones de dispersión* o *funciones hash*.

> Una tabla de dispersión consta de un *array* (vector), donde se almacenan los registros o elementos, y de una función *hash,* que transforma el campo clave elegido en el rango entero del *array*.

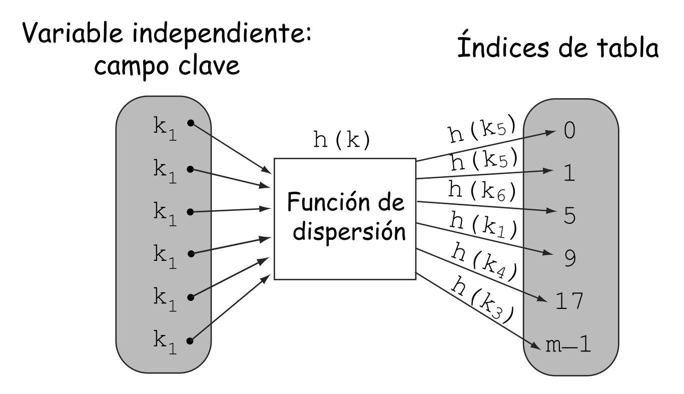


Figura 12.1 Función de dispersión

12.1.2. Operaciones de tablas de dispersión

Las tablas dispersas implementan eficientemente los tipos de datos denominados *Diccionarios*. Un diccionario asocia una *clave* con un *valor*. Por ejemplo, el diccionario de inglés asocia una palabra con la traducción en inglés. La operación *búsqueda* de una palabra se realiza frecuentemente; dada una palabra, se localiza su posición y se obtiene la traducción correspondiente (el *valor asociado*).

La primera operación relativa a una tabla dispersa consiste en su creación, esto es, dar de alta elementos. La operación de *insertar* añade un elemento en la posición que le corresponde según la clave del elemento. De igual manera, también pueden darse de baja elementos: la operación *eliminar* extrae un elemento de la tabla que se encuentra en la posición que le corresponde según su clave.

En las tablas dispersas no se utiliza directamente la clave para indexar, el índice se calcula con una función matemática, *función hash h(x),* que tiene como argumento la clave del elemento y devuelve una dirección o índice en el rango de la tabla. Según esto, considerando *h(x)* la función *hash,* se pueden especificar las operaciones del tipo *Tabla dispersa:*

Buscar(Tabla T, clave x)

```
devuelve el elemento de la tabla T[h(x)]
```

Insertar(Tabla T, elemento k)

```
añade el elemento k, T[h(clave(k)] ← k
```

Eliminar(Tabla T, clave x)

```
retira de la tabla el elemento con clave x, T[h(x)] ← LIBRE
```

La ventaja de utilizar tablas de dispersión radica en la eficiencia de estas operaciones. Si la función *hash* es de complejidad constante, la complejidad de cada una de las tres operaciones también es constante, $O(1)$. Un problema potencial de la función de dispersión es el de las colisiones, esto es dadas dos claves distintas x_i, x_j se obtenga la misma dirección o índice: $h(x_i) = h(x_j)$. La operación de *insertar* tiene que incorporar el proceso de *resolución de colisiones,* no pueden estar dos elementos en la misma posición. De igual forma, los procesos de *búsqueda* y *eliminación* también quedan afectados por la resolución de colisiones.

Las tablas dispersas se diseñan considerando el problema de las colisiones. Siempre se reservan mas posiciones de memoria, m, que elementos previstos a almacenar, n. Cuantas más posiciones haya, menor es el riesgo de colisiones, pero mas *huecos* libres quedan (memoria desaprovechada).

El parámetro que mide la proporción entre el número de elementos almacenados, n, en una tabla dispersa y el tamaño de la tabla, m, se denomina factor de carga, $\lambda = n/m$. Se recomienda elegir m de tal forma que el factor de carga sea $\lambda \leq 0.8$.

12.2. FUNCIONES DE DISPERSIÓN

Una *función de dispersión* convierte el dato considerado campo clave (tipo entero o cadena de caracteres) en un índice dentro del rango de definición del *array* o vector que almacena los elementos, de tal forma que sea adecuado para indexar el *array.*

La idea que subyace es utilizar la clave de un elemento para determinar su dirección o posición en un almacenamiento secuencial, pero sin desperdiciar mucho espacio. Para ello, se realiza una transformación, mediante una *función hash*, del conjunto *K* de claves sobre el conjunto *L* de direcciones de memoria.

$$h(x): K \rightarrow L$$

Ésta es la función de direccionamiento *hash* o función de dispersión. Si *x* es una clave, entonces *h(x)* se denomina direccionamiento *hash* de la clave *x,* y además es el índice de la tabla donde se guardará el registro con esa clave. Así, si la tabla tiene un tamaño de tamTabla = 199, la *función hash* que se elija tiene generar índices en el rango 0 ... tamTabla-1. Si la clave es un

entero, por ejemplo, el número de serie de un artículo (hasta 6 dígitos), y se dispone de una tabla de tamTabla elementos, la función de direccionamiento tiene que ser capaz de transformar valores pertenecientes al rango 0 ... 999999 en valores pertenecientes al subrango 0 ... tamTabla-1. La clave elegida también puede ser una cadena de caracteres, en ese caso se hace una transformación previa a valor entero. La función *hash* más simple utiliza el operador *módulo (resto entero)*,

x % tamTabla = genera un número entero de 0 a tamTabla-1

Es necesario valorar el hecho de que la *función hash*, h(x), no genere valores distintos, es posible (según la función elegida) que dos claves diferentes, c1 y c2, den la misma dirección, h(c1) = h(c2). Entonces se produce el fenómeno de la *colisión,* y se debe usar algún método para resolverla. Por tanto, el estudio del *direccionamiento hash* implica dos hechos: la elección de *funciones hash* y *resolución de colisiones*.

Existe un número considerable de *funciones hash*. Dos criterios deben considerarse a la hora de seleccionar una función. En primer lugar, que la función, h(x), sea fácil de evaluar (lo que dependerá del campo clave) y que su tiempo de ejecución sea mínimo, de complejidad constante, O(1). En segundo lugar, h(x) debe distribuir uniformemente las direcciones sobre el conjunto L (direcciones de memoria), de forma que se minimice el número de *colisiones*. Nunca existirá una garantía plena de que no haya colisiones, y más sin conocer de antemano las claves y las direcciones. La experiencia enseña que siempre habrá que preparar la *resolución de colisiones* para cuando éstas se produzcan.

Algunas de las *funciones hash* de cálculo más fácil y rápido se exponen en las secciones siguientes. Para todas ellas se sigue el criterio de considerar x una clave cualquiera, m el tamaño de la tabla y, por tanto, los índices de la tabla varían de 0 a m-1; y, por último, el número de elementos es n.

12.2.1. Aritmética modular

Una *función de dispersión* que utiliza la *aritmética modular* genera valores dispersos calculando el resto de la división entera entre la clave x y el tamaño de la tabla m.

h(x)= x *modulo* m

Normalmente, la clave asociada con un elemento es de tipo entero. A veces, los valores de la clave no son enteros; entonces, previamente hay que transformar la clave a un valor entero. Por ejemplo, si la clave es una cadena de caracteres, se puede transformar considerando el valor ASCII de cada carácter como si fuera un dígito entero de base 128.

La operación *resto(módulo)* genera un numero entre 0 y m-1 cuando el segundo operando es m. Por tanto, esta *función de dispersión* proporciona valores enteros dentro del rango 0 ... m-1.

Con el fin de que esta función disperse lo más uniformemente posible, es necesario tener ciertas precauciones con la elección del tamaño de la tabla, m. Así, no es recomendable escoger el valor de m múltiplo de 2 ni tampoco de 10. Si $m = 2^j$, entonces la distribución de h(x) se basa únicamente en los j dígitos menos significativos; y si $m = 10^j$, ocurre lo mismo. En estos casos, la distribución de las claves se basa sólo en una parte, los j dígitos menos significativos, de la información que suministra la clave, y esto sesgará la dispersión hacia ciertos valores o índices de la tabla. Si, por ejemplo, el tamaño elegido para la tabla es m = 100 (10^2), entonces h(128) = h(228) = h(628) = 28.

Las elecciones recomendadas del tamaño de la tabla, `m`, son números primos mayores, aunque cercano al número de elementos, `n` , que tiene previsto que almacene la tabla.

Ejemplo 12.1

Considerar una aplicación en la que se deben almacenar n = 900 registros. El campo clave elegido es el número de identificación. Elegir el tamaño de la tabla de dispersión y calcular la posición que ocupan los elementos cuyo número de identificación es:

```
245643   245981    257135
```

Una buena elección de `m`, en este supuesto, es `997` al ser un número primo próximo y tener como factor de carga `(n/m)` aproximadamente `0.8` cuando se hayan guardado todos los elementos.

Teniendo en cuenta el valor de `m`, se aplica la función *hash* de aritmética modular y se obtienen estas direcciones:

```
h(245643)  =  245643 modulo 997 =  381
h(245981)  =  245981 modulo 997 =  719
h(257135)  =  257135 modulo 997 =  906
```

12.2.2. Plegamiento

La técnica del plegamiento se utiliza cuando el valor entero del campo clave elegido es demasiado grande, pudiendo ocurrir que no pueda ser almacenado en memoria. Consiste en partir la clave x en varias partes: $x_1, x_2, x_3 \ldots x_n$. La combinación de las partes de un modo conveniente (a menudo sumando las partes) da como resultado la dirección del registro. Cada parte x_i, con a lo sumo la excepción de la última, tiene el mismo número de dígitos que la dirección especificada.

La función *hash* se define de la siguiente forma:

$$h(x) = x_1 + x_2 + x_3 + \ldots + x_r$$

La operación que se realiza para el cálculo de la función *hash* desprecia los dígitos mas significativos obtenidos del acarreo.

> **Nota**
>
> La técnica de plegar la clave de dispersión se utiliza a menudo para transformar una clave muy *grande* en otra más *pequeña* y, a continuación, aplicar la función *hash* de aritmética modular.

Ejemplo 12.2

Los registros de pasajeros de un tren de largo recorrido se identifican por un campo de 6 dígitos, que se va a utilizar como clave para crear una tabla dispersa de m = 1.000 posiciones (rango de 0 a 999). La función de dispersión utiliza la técnica de plegamiento de tal forma que parte la clave, 6 dígitos, en dos grupos de tres dígitos y, a continuación, se suman los valores de cada grupo.

Aplicar esta función a los siguientes registros:

```
245643     245981    257135
```

Al dividir la primera clave, `245643` en dos grupos de tres dígitos se obtiene `245` y `643`. La dirección obtenida es:

```
h(245643) = 245 + 643 = 888
```

Para las otras claves:

`h(245981) = 245 + 981 = 1226 = 226` (se ignora el acarreo `1`)
`h(257135) = 257 + 135 = 392`

A veces se determina la inversa de las partes pares, x_2, x_4 ... antes de sumarlas, con el fin de conseguir mayor dispersión. La inversa en el sentido de invertir el orden de los dígitos. Así, con las mismas claves se obtienen las direcciones:

```
h(245643) = 245 + 346 = 591
h(245981) = 245 + 189 = 434
h(257135) = 257 + 531 = 788
```

12.2.3. Mitad del cuadrado

Esta técnica de obtener direcciones dispersas consiste, en primer lugar, en calcular el cuadrado de la clave `x` y, a continuación, extraer del resultado, x^2, los dígitos que se encuentran en ciertas posiciones. El número de dígitos a extraer depende del rango de dispersión que se quiera obtener. Así, si el rango es de `0 ... 999` se extraen tres dígitos, siempre los que están en las mismas posiciones.

Un problema potencial, al calcular x^2, es que sea demasiado grande y *exceda* el máximo entero. Es importante, al aplicar este método de dispersión, utilizar siempre las mismas posiciones de extracción para todas las claves.

Ejemplo 12.3

Aplicar el método de mitad del cuadrado a los mismos los registros del Ejemplo 12.2.

Una vez elevado al cuadrado el valor de la clave, se eligen los dígitos que se encuentran en las posiciones 4, 5 y 6 por la derecha. El valor de esa secuencia es la dirección obtenida al aplicar este método de dispersión.

Para `245643, h(245643) = 483;` *paso a paso:*
`245643` → 245643^2 → `60340483449` → (dígitos 4, 5 y 6 por la derecha) `483`

Para `245981, h(245981) = 652;` *paso a paso:*
`245981` → 245981^2 → `60506652361` → (dígitos 4, 5 y 6 por la derecha) `652`

Para `257135, h(257135) = 408;` *paso a paso:*
`257135` → 257135^2 → `66118408225` → (dígitos 4, 5 y 6 por la derecha) `408`

12.2.4. Método de la multiplicación

La dispersión de una clave utilizando el *método de la multiplicación* genera direcciones en tres pasos. Primero, multiplica la clave `x` por una constante real, `R`, comprendida entre `0` y `1` (`0. < R < 1.0`). En segundo lugar, determina la parte decimal, `d`, del número obtenido en la multiplicación,

R*x. Y, por último, multiplica el tamaño de la tabla, m, por d y al truncarse el resultado se obtiene un número entero en el rango 0 ... m-1 que será la dirección dispersa.

```
1. R * x
2. d = R * x - ParteEntera(R * x)
3. h(x) = ParteEntera(m * d)
```

Una elección de la constante R es la inversa de la razón áurea, R = 0.6180334. Por ejemplo, con la clave x = 245981, m = 1000, la dirección que se obtiene:

```
1. R * x: 0.6180334 * 245981 → 152024.4738
2. d: 152024.4738 - ParteEntera(152024.4738) → 0.4738
3. h(245981): 1000 * 0.4738 → ParteEntera (473.8) → 473
```

Este método de obtener direcciones dispersas tiene dos características importantes. La primera, dos claves con los dígitos permutados, no tienen mayor probabilidad de generar una colisión (igual dirección) que dos claves cualesquiera. La segunda, dos valores de claves numéricamente muy próximas, generan direcciones dispersas que pueden estar muy separadas.

Considérese, por ejemplo, la clave x = 245982 para el mismo valor de R = 0.6180334 y m = 1000, la dispersión sería:

```
1. R * x: 0.6180334 * 245982 → 152025.0918
2. d: 152025.0918 - ParteEntera(152025.0918) → 0.0918
3. h(245982): 1000 * 0.0918 → ParteEntera (91.8) → 91
```

La dispersión obtenida para 245891, h(245981) = 473 muy alejada de 91.

Ejercicio 12.1

Los registros que representan los objetos de una perfumería se van a guardar en una tabla dispersa de m = 1.024 posiciones. El campo clave es una cadena de caracteres, de la que se toman los 10 primeros. Se decide aplicar el método de la multiplicación como función de dispersión. Con este supuesto, codificar la función de dispersión y mostrar 10 direcciones dispersas.

Para generar la dispersión de las claves, primero se transforma la cadena, que es el campo clave, en un valor entero; una vez hecha la transformación, se aplica el método de la multiplicación.

La transformación de la cadena se realiza considerando que es una secuencia de valores numéricos de base 27. Así, por ejemplo, la cadena 'RIO' se transforma en:

$$'R'*27^2 + 'I'*27^1 + 'O'*27^0$$

El valor entero de cada carácter es su ordinal en el código ASCII (subconjunto del conjunto de caracteres *Unicode*). El tipo de dato char ocupa dos bytes de memoria y se representa como un valor entero que es, precisamente, el ordinal. Para obtener el valor entero de un carácter, simplemente se hace una conversión a int; por ejemplo, (int)'a' es el valor entero 97.

La transformación da lugar a valores que sobrepasan el máximo entero (incluso con enteros largos), generando números negativos. No es un problema, ya que sencillamente se cambia de signo.

Para generar las 10 direcciones dispersas y probar la función, el programa tiene como entrada cada una de las 10 cadenas; se invoca al método que implementa la función de dispersión y se escribe la dirección.

```
/*
        función hash: método de la multiplicación
   Las claves son cadenas de caracteres, primero se transforma
   a valor entero. A continuación se aplica el método de multiplicación
   */

   import java.io.*;

public class DispersionHash
{
   static final int M = 1024;
   static final double R = 0.618034;

   static long transformaClave(String clave)
   {
      long d;

      d = 0;
      for (int j = 0; j < Math.min(clave.length(),10); j++)
      {
         d = d * 27 + (int)clave.charAt(j);
      }
      /*
         Para un valor mayor que el máximo entero genera un
         numero negativo.
      */
         if (d < 0) d = -d;
         return d;
      }
   static int dispersion(long x)
   {
   double t;
   int v;
   t = R * x - Math.floor(R * x); // parte decimal
   v = (int) (M * t);
   return v;
   }
   public static void main(String[]a) throws IOException
   {
      String clave;
      long valor;
      BufferedReader entrada = new BufferedReader(
                     new InputStreamReader(System.in));

      for (int k = 1; k <= 10; k++)
      {
         System.out.print("\nClave a dispersar: ");
         clave = entrada.readLine();
         valor = transformaClave(clave);
         valor = dispersion(valor);
         System.out.println("Dispersion de la clave " +
                                   clave + " \t " + valor);
      }
   }
}
```

12.3. COLISIONES Y RESOLUCIÓN DE COLISIONES

La función de dispersión elegida `h(x)` puede generar la misma posición al aplicarla a las claves de dos o más registros diferentes; esto es, obtener la misma posición de la tabla en la que ubicar dos registros. Si ocurre, se produce una **colisión** que es preciso resolver para que los registros ocupen diferentes posiciones.

Una función *hash* ideal, `h(x)`, debe generar direcciones distintas para dos claves distintas. No siempre es así, no siempre proporciona direcciones distintas; en ocasiones, ocurre que dadas dos claves diferentes $x_1, x_2 \Rightarrow h(x_1) = h(x_2)$. Este hecho es conocido como colisión, es evidente que el diseño una tabla dispersa debe proporcionar métodos de *resolución de colisiones*.

> ***Recordar***
>
> Hay dos cuestiones que se deben considerar a la hora de diseñar una tabla *hash*. Primero, seleccionar una buena función *hash,* que *disperse lo mas uniformemente*. Segundo, seleccionar un *método para resolver colisiones.*

Considérese, por ejemplo, una comunidad de 88 vecinos, cada uno con muchos campos de información relevantes: nombre, edad... de los que se elige como campo clave el *DNI*. El tamaño de la tabla va a ser 101 (número primo mayor que 88), por consiguiente, habrá 101 posibles direcciones. Aplicando la función *hash* del *módulo*, las claves 123445678, 123445880 proporcionan las direcciones:

```
h(123445678) = 123445678 modulo 101 = 44
h(123445880) = 123445880 modulo 101 = 44
```

Para dos claves distintas a las que se aplica la función *hash* del módulo, si se obtienen dos direcciones iguales, se dice que las claves han colisionado.

Es importante tener en cuenta que la *resolución de colisiones,* en una tabla dispersa, afecta directamente a la eficiencia de las operaciones básicas sobre la tabla: *insertar, buscar* y *eliminar.*

Se consideran dos modelos para resolver colisiones: la *exploración* de direcciones y el *direccionamiento enlazado*. En las siguientes secciones se muestran los diversos métodos de resolución.

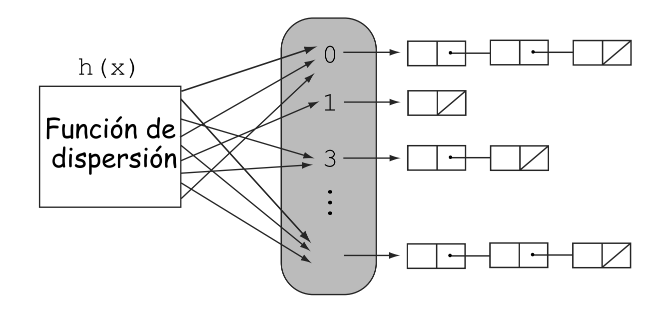


Figura 12.2 Resolución de colisiones por dispersión abierta (enlazada)

12.4. EXPLORACIÓN DE DIRECCIONES

Los diversos métodos de exploración se utilizan cuando todos los elementos, colisionados o no, se almacenan en la misma tabla. Las colisiones se resuelven explorando consecutivamente en una secuencia de direcciones hasta que se encuentra una posición *libre* (un *hueco*) en la operación de insertar o se encuentra el elemento buscado en las operaciones *buscar* y *eliminar*.

Es importante, al diseñar una tabla dispersa basada en la resolución de colisiones en la *exploración de una secuencia,* inicializar todas las posiciones de la tabla a un valor que indique vacío, por ejemplo, `null` o cualquier parámetro que indique posición vacía. Al insertar un elemento, si se produce una colisión, la secuencia de exploración termina cuando se encuentra una dirección de la secuencia vacía.

Al buscar un elemento, se obtiene la dirección dispersa según su clave. A partir de esa dirección es posible que se necesite explorar la secuencia de posiciones hasta encontrar la clave buscada. En la operación de eliminar, una vez encontrada la clave, se indica con un parámetro el estado borrado. Dependiendo de la aplicación, una posición eliminada puede utilizarse, posteriormente, para una inserción.

Nota

La secuencia de posiciones (índices) a la que da lugar un método de exploración tiene que ser la misma, independiente de la operación que realiza la tabla dispersa. Es importante marcar con un parámetro que una posición de la tabla está vacía.

12.4.1. Exploración lineal

Es la forma más primaria y simple de resolver una colisión entre claves, al aplicar una función de dispersión. Supóngase que se tiene un elemento de clave `x`, la dirección que devuelve la función `h(x) = p`, si esta posición ya está ocupada por otro elemento se ha producido una colisión. La forma de resolver está colisión con *exploración lineal* consiste en buscar la primera posición disponible que siga a `p`. La secuencia de exploración que se genera es lineal: `p, p+1, p+2, ... m-1, 0, 1, ...` y así consecutivamente hasta encontrar una posición vacía. La tabla se ha de considerar *circular,* de tal forma que siendo `m-1` la última posición, la siguiente es la posición `0`.

Ejemplo 12.4

Se tienen 9 elementos cuyas claves simbólicas son `x1, x2, x3, x4, x5, x6, x7, x8` *y* `x9`. *Para cada unom la función de dispersión genera las siguientes direcciones:*

```
Elemento: x1   x2   x3   x4   x5   x6   x7   x8   x9
h(x) :     5    8    11   9    5    7    8    6   14
```

Entonces, las posiciones de almacenamiento en la tabla, aplicando exploración lineal son:

```
Elemento: x1   x2   x3   x4   x5   x6   x7   x8   x9
Posición:  5    8   11    9    6    7   10   12   14
```

La primera colisión se produce al almacenar el elemento x5, según la función *hash,* le corresponde la posición 5, que está ocupada. La siguiente posición esta libre, y a ella se asigna el elemento. Al elemento x8 le corresponde la dirección dispersa 6, sin embargo, esa posición ya está ocupada por x5 debido a una colisión previa. La siguiente posición libre es la 12, dirección que le corresponde al resolver la colisión con el método de exploración lineal.

Las operaciones *buscar* y *eliminar* un elemento en una tabla dispersa con exploración lineal siguen los mismos pasos que la operación de *insertar* (Ejemplo 12.4). La búsqueda comienza en la posición que devuelve la función *hash,* h(clave). Si la clave en esa posición es la clave de búsqueda, la operación ha tenido éxito; en caso contrario, la exploración continúa linealmente en las siguientes posiciones hasta encontrar la clave o bien decidir que no existe en la tabla, lo que ocurre si una posición está vacía. Por ejemplo, para buscar el elemento x8 en la tabla del ejemplo anterior, se examinan las posiciones 6, 7, 8, 9, 10 y 12.

La exploración lineal es sencilla de implementar, tiene un inconveniente importante, el agrupamiento de elementos en la tabla. Cuando el factor de ocupación se acerca a 0.5, el agrupamiento de elementos en posiciones adyacentes es notable.

Análisis de la exploración lineal

La eficiencia de una función *hash,* junto al método de resolución de colisiones, supone analizar la operación de *insertar* y la operación de *buscar* (*eliminar* supone una búsqueda). El factor de carga de la tabla, λ, es determinante a la hora de determinar la eficiencia de las operaciones.

La eficiencia de la inserción se suele medir como *el número medio de posiciones examinadas*. En la exploración lineal, esta magnitud se aproxima a:

$$\frac{1}{2}\left(1+\frac{1}{(1-\lambda)^2}\right)$$

Al analizar la operación de búsqueda de un elemento, se tiene en cuenta que tenga éxito, o bien que sea una búsqueda sin éxito (que no se encuentre). Por esa razón, la eficiencia de la operación buscar se expresa mediante dos parámetros:

$S(\lambda)$ = número medio de comparaciones para una búsqueda con éxito.
$U(\lambda)$ = número medio de comparaciones para una búsqueda sin éxito.

Se puede demostrar que, para la exploración lineal, estos parámetros serían:

$$S(\lambda) = \frac{1}{2}\left(1+\frac{1}{1-\lambda}\right) \quad \text{y} \quad U(\lambda) = \frac{1}{2}\left(1+\frac{1}{(1-\lambda)^2}\right)$$

Nota de programación

La exploración lineal es fácil de implementar en cualquier lenguaje de programación. Tiene como principal inconveniente, cuando el factor de carga supera el 50%, el agrupamiento de elementos en posiciones contiguas. Situar los elementos en posiciones contiguas aumenta el tiempo medio de la operación de búsqueda.

12.4.2. Exploración cuadrática

La resolución de colisiones con la exploración lineal provoca que se agrupen los elementos de la tabla según se va acercando el factor de carga a `0.5`. Una alternativa para evitar la agrupación es la *exploración cuadrática*. Suponiendo que a un elemento con clave `x` le corresponde la dirección `p` y que la posición de la tabla indexada por `p` está *ocupada,* el método de exploración o prueba cuadrática busca en las direcciones `p, p+1, p+4, p+9, ...` $p+i^2$, considerando la tabla como un *array* circular. El nombre de *cuadrática* para esta forma de explorar se debe al desplazamiento relativo, i^2 para valores de `i= 1, 2, 3...`

Ejemplo 12.5

Se dispone de `9` *elementos cuyas claves simbólicas son:* `x1, x2, x3, x4, x5, x6, x7, x8` *y* `x9`. *El tamaño de la tabla donde se guardan es* `17` *(número primo), rango de* `0` *a* `16`. *Para cada uno de los elementos la función de dispersión genera las direcciones:*

```
Elemento: x1  x2  x3  x4  x5  x6  x7  x8  x9
h(x):      5   8  10   8  5   11   6   7   7
```

Las posiciones de almacenamiento en la tabla, aplicando la exploración cuadrática:

```
Elemento: x1  x2  x3  x4  x5  x6  x7  x8  x9
Posición:  5   8  10   9   6  11   7  16  15
```

La primera colisión se produce al almacenar el elemento `x4`. Según la función *hash,* le corresponde la posición `8,` que está ocupada. Si se considera la secuencia de exploración cuadrática: `8, 8+1, 8+4, ...` como la posición `9` está libre ,en ella se asigna el elemento.

Al elemento `x5` le corresponde la dirección dispersa `5`, sin embargo, hay una colisión con el elemento `x1;` la secuencia de exploración para esta clave es `5, 5+1, 5+4, ...`, como la posición `6` está libre, se asigna el elemento.

La siguiente colisión se da con `x7`, le corresponde la dirección dispersa `6` y esa posición está ya ocupada debido a una colisión previa; la secuencia de exploración para esta clave es, `6, 6+1, 6+4, ...`, la posición `7` está libre y se asigna el elemento.

Al elemento `x8` le corresponde la dirección `7` ya ocupada. La secuencia de exploración, `7, 7+1, 7+4, 7+9, ...`, resulta que la posición `16` está libre, en ella se asigna el elemento.

Por último, al elemento `x9` le corresponde la dirección dispersa `7,` ya asignada. La secuencia de exploración cuadrática en la que se prueba hasta encontrar una posición libre es `7, 7+1, 7+4, 7+9, 7+16(6), 7+25 ...` Resulta que ahora hay que tratar la secuencia en forma circular al estar ocupadas las cuatro primeras posiciones. El siguiente valor de la secuencia es `23`, que se corresponde con `6` *(teoría de restos)*, también está ocupado. A continuación, se prueba la dirección `32`, que se corresponde con `15` por la *teoría de restos,* y al estar libre es la posición donde se asigna el elemento.

Análisis de la exploración cuadrática

Este método de resolver colisiones reduce el agrupamiento que produce la exploración lineal. Sin embargo, si no se elige convenientemente el tamaño de la tabla, no se puede asegurar que se prueben todas las posiciones de la tabla.

Se puede demostrar que si el tamaño de la tabla es un número primo y el factor de carga no alcanza el `50%,` todas las *pruebas* que se realicen con la secuencia $p+i^2$ se hacen sobre posiciones de la tabla distintas y siempre se podrá insertar.

12.4.3. Doble dirección dispersa

Este método de resolución de colisiones utiliza una segunda función *hash*. Se tiene una función *hash* principal, `h(x)`, y otra función secundaria, `h'(x)`. El primer intento de insertar, o de buscar, un nuevo elemento inspecciona la posición `h(x) = p`, si hay una colisión, se obtiene un segundo desplazamiento con otra función *hash*, `h'(x) = p'`. Entonces, la secuencia de exploración `p, p+p', p+2p', p+3p' ...` se inspecciona hasta encontrar una posición libre, para insertar, el elemento buscado.

12.5. REALIZACION DE UNA TABLA DISPERSA

A continuación se implementa una tabla dispersa para almacenar un conjunto de elementos: *Casas Rurales* de la comarca *La Alcarria*. La resolución de colisiones se realiza formando una secuencia de posiciones aplicando el método de *exploración cuadrática*.

Los elementos de la tabla son objetos con los siguientes datos: *población, dirección, numHabitacion, precio* por día y *código* de identificación. El código, normalmente 5 caracteres, tiene una relación biunívoca con la *Casa Rural*; por ello, se elige como campo clave. Los campos *población* y *dirección* son de tipo cadena; y *numHabitacion*, *precio* de tipo `int` y `double` respectivamente.

La clase `TablaDispersa` consta de un *array* de referencias a los objetos `CasaRural`. El tamaño de la tabla está en función del número de *Casas* conocido, `50`. Entonces, el tamaño elegido es el número primo `101`. Cada posición de la tabla contiene `null` o bien la referencia a un objeto *Casa Rural.* Se ha tomado la decisión de que los elementos dados de baja permanezcan en la tabla (para no perder información histórica); por ello, se añade el atributo `esAlta`, que si está activo *(true)* indica que es un *alta*; en caso contrario *(false)* se dio de *baja*. El número de elementos que hay en la tabla, incluyendo las bajas, se almacena en la variable *`numElementos`;* además, se añade la variable `factorCarga`, de tal forma que cuando se alcance el `0.5` se pueda generar un *aviso*.

12.5.1. Declaración de la clase TablaDispersa

El constructor de la clase crea el *array,* inicializa a `null` cada posición y a cero `numElementos` y `factorCarga`. En primer lugar, se declara la clase `CasaRural` y, a continuación, la clase que encapsula la tabla dispersa de casa rurales.

```
package CasasAlcarria;
import java.io.*;

public class CasaRural
{
   private String codigo;
   private String poblacion;
   private String direccion;
   private int numHabitacion = 0;
   private double precio = 0.0;
   boolean esAlta;
   public CasaRural()
   {
      esAlta = true;
      asigna();
   }
```

```
   public void asigna()
   {
      BufferedReader entrada = new BufferedReader(
                  new InputStreamReader(System.in));
      try {
         System.out.print("\n Codigo (10 caracteres): ");
         codigo = entrada.readLine();
         System.out.print("\n Población: ");
         poblacion = entrada.readLine();
         System.out.print("\n Dirección: ");
         direccion = entrada.readLine();
         System.out.print("\n Número de habitaciones: ");
         numHabitacion = Integer.parseInt(entrada.readLine());
         System.out.print("\n Precio por día de estancia: ");
         precio = (new Double(entrada.readLine())).doubleValue();
      }
      catch (IOException e)
      {
         System.out.println(" Excepcion en la entrada de datos  " +
                        e.getMessage()+ " . No se da de alta");
         esAlta = false;
      }
   }
   public String elCodigo()
   {
      return codigo;
   }
   public void muestra()
   {
      System.out.println("\n Casa Rural " + codigo);
      System.out.println("Población: " + poblacion);
      System.out.println("Dirección: " + direccion);
      System.out.println("Precio por día: " + precio);
   }
}
```

Las operaciones de la clase `TablaDispersa` se realizan en los siguientes apartados:

```
package CasasAlcarria;

public class TablaDispersa
{
   static final int TAMTABLA = 101;
   private int numElementos;
   private double factorCarga;
   private CasaRural [] tabla;

   //...
```

12.5.2. Inicialización de la tabla dispersa

El constructor de `TablaDispersa` crea el *array* con el tamaño especificado e inicializa a `null` cada posición. Los atributos `numElementos` y `factorCarga` se inicializan a `0`.

```
public TablaDispersa()
{
```

```
      tabla = new CasaRural[TAMTABLA];
      for(int j = 0; j < TAMTABLA ; j++)
         tabla[j] = null;
      numElementos = 0;
      factorCarga = 0.0;
   }
```

12.5.3. Posición de un elemento

Al añadir un nuevo elemento a la tabla, o bien al buscar el elemento, es necesario determinar la posición en la tabla. Se utiliza la función *aritmética modular* para obtener la dirección dispersa, a partir de la cual se forma una secuencia de exploración cuadrática en la que se busca la primera posición libre (posición a `null`). Al ser la clave de *dispersión* de tipo cadena, primero se convierte ésta a un valor entero. El método `transforma()` realiza la conversión de igual forma que en el Ejercicio 12.1.

Se prueba secuencialmente cada posición que determina el método de *exploración cuadrática* hasta encontrar una posición vacía `(null)`, o bien, una posición con la misma clave. El método `direccion()` gestiona estas acciones, devuelve la posición o índice de la tabla.

```
   public int direccion(String clave)
   {
      int i = 0, p;
      long d;
      d = transformaCadena(clave);
      // aplica aritmética modular para obtener dirección base
      p =  (int)(d % TAMTABLA);
      // bucle de exploración
      while (tabla[p]!= null   &&
                          !tabla[p].elCodigo().equals(clave))
      {
         i++;
         p = p + i*i;
         p = p % TAMTABLA;  // considera el array como circular
      }
      return p;
   }

   long transformaCadena(String c)
   {
      long d;

      d = 0;
      for (int j = 0; j < Math.min(10,c.length()); j++)
      {
         d = d * 27 + (int)c.charAt(j);
      }
      if (d < 0) d = -d;
      return d;
   }
```

12.5.4. Insertar un elemento en la tabla

Para incorporar un nuevo elemento a la tabla, primero se busca la posición que debe ocupar, el método `direccion()`devuelve la posición buscada. La operación no tiene en cuenta que en

esa posición haya un elemento, si es así se *sobrescribe*. El número de entradas de la tabla se incrementa y se hace un nuevo cálculo del *factor de carga*. La única acción que se hace con el factor de carga es escribir un mensaje de aviso, si este supera el `50%`.

```
public void insertar(CasaRural r)
{
   int posicion;
   posicion = direccion(r.elCodigo());
   tabla[posicion] = r;
   numElementos++;
   factorCarga = (double)(numElementos)/TAMTABLA;
   if (factorCarga > 0.5)
      System.out.println("\n!! Factor de carga supera el 50%.!!"
                         + " Conviene aumentar el tamaño." );
}
```

12.5.5. Búsqueda de un elemento

La operación busca un elemento en la tabla a partir de la dirección dispersa correspondiente a la clave. El método `buscar()` devuelve una referencia al elemento si se encuentra en la tabla; de no encontrarse, o bien si fue dado de baja, devuelve `null`.

```
public CasaRural buscar(String clave)
{
   CasaRural pr;
   int posicion;
   posicion = direccion(clave);
   pr = tabla[posicion];
   if (pr != null)
      if (! pr.esAlta) pr = null;
   return pr;
}
```

12.5.6. Dar de baja un elemento

Para dar de baja un elemento, se siguen los mismos paso que en la operación de búsqueda. Primero se determina la posición del elemento en la tabla, llamando al método `direccion()`. A continuación, si en la posición hay un elemento, simplemente se pone a `false` el campo `esAlta`.

```
public void eliminar(String clave)
{
   int posicion;
   posicion = direccion(clave);
   if (tabla[posicion] != null)
      tabla[posicion].esAlta = false;
}
```

12.6. DIRECCIONAMIENTO ENLAZADO

En la sección anterior se ha desarrollado una estrategia para la resolución de colisiones que, de una manera o de otra, forma una secuencia de posiciones a explorar. Una alternativa a la secuencia de exploración es el direccionamiento *(hashing)* enlazado. Se basa en utilizar listas enlazadas (cadenas de elementos), de tal forma que en cada lista se colocan los elementos que tienen la

misma dirección *hash*. Todos los elementos que *colisionan*: `h(x1) = h(x2) = h(x3) ...` van a estar ubicados en la misma lista enlazada.

La Figura 12.3 muestra la estructura de datos básica para este método de dispersión. La idea fundamental es la siguiente: si se ha elegido una función *hash* que genera direcciones en el rango `0` a `m-1`, se debe crear una tabla de `m` posiciones, indexada de `0` a `m-1`; cada posición de la tabla contendrá la dirección (referencia) de acceso a su correspondiente lista enlazada.

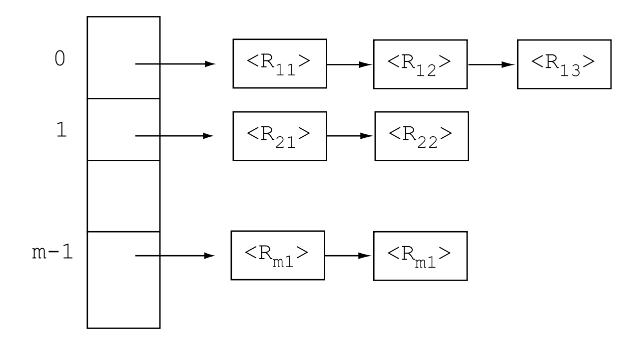


Figura 12.3 Tabla de dispersión enlazada con rango `0..m-1` direcciones.

La estructura de datos empleada para la implementación de una tabla dispersa con direccionamiento enlazado es un *array* de listas enlazadas. Cada posición `i` es una referencia a la lista enlazada, sus nodos son elementos de la tabla dispersa cuyo direccionamiento *hash,* obtenido con la función *hash* elegida, es `i`. Ahora, los elementos de la tabla dispersa se agrupan en listas enlazadas, por consiguiente es necesario que dispongan de un campo adicional para poder enlazar con el *siguiente* elemento. La declaración de clase *Elemento* es:

```
class Elemento
{
   //
   < atributos según los datos del elemento a representar >
   //

   Elemento sgte;

};
```

La declaración de la *Tabla* se hace con un *array* de referencias a `Elemento`. También se puede optar por utilizar la clase `Vector`, (paquete `java.util`) para almacenar cualquier tipo de objeto, aunque presenta el inconveniente, al recuperar elementos, de que es necesario hacer la conversión de `Object` al tipo `Elemento`. Se ha elegido la estructura de *array* para evitar realizar conversiones, y para una mejor comprensión de las operaciones realizadas. La constante `M`, en esta implementación, es el número de posibles direcciones dispersas y, en consecuencia, el número de listas enlazadas.

```
public class TablaDispersaEnlazada
{
   static final int M = 101;
   private Elemento [] tabla;
   private int numElementos;
   // operaciones
```

12.6.1. Operaciones de la tabla dispersa enlazada

Las operaciones fundamentales que se realizan con tablas dispersas, *insertar, buscar* y *eliminar,* con esta forma de implementación se convierten, una vez obtenido el índice de la tabla con la función *hash,* en operaciones sobre listas enlazadas.

Para añadir un elemento a la tabla cuya clave es `x`, se inserta en la lista de índice `h(x)`. Puede ocurrir que la lista esté vacía, no se almacenó ningún elemento con esa dirección dispersa, o bien que ya tenga elementos. En cualquier caso, la inserción en la lista se hace como primer elemento; de esa forma, el tiempo de ejecución es de complejidad constante; por esa razón se puede afirmar que la complejidad de la operación *inserción* es `O(1)`.

Ahora, un elemento de la tabla puede eliminarse *físicamente,* no es necesario marcar el elemento como *no activo*, como ocurre con las *secuencias de exploración*. Los pasos a seguir son: primero se obtiene el índice de la tabla, `h(x)`, y a continuación se aplica la operación *eliminar* un nodo de una lista enlazada.

La búsqueda de un elemento sigue los mismos pasos: se obtiene el índice de la tabla mediante la función *hash* y, a continuación, se aplica la operación *búsqueda* en listas enlazadas.

12.6.2. Análisis del direccionamiento enlazado

El planteamiento de *hashing encadenado* permite acceder directamente a la lista enlazada en la que se busca un elemento. La búsqueda es una operación lineal, en el *peor de los casos* recorre todos los nodos de la lista; a pesar de ello, las listas han de tener un número de nodos suficientemente pequeño, y así el tiempo de ejecución será reducido.

Considerando el factor de carga, λ, la longitud media de cada lista enlazada (número de nodos) es, precisamente, λ. Esto permite concluir que el número medio de nodos visitados en una búsqueda sin éxito es λ, y el promedio de nodos visitados en una búsqueda con éxito es $1 + \frac{1}{2}\lambda$.

El factor de carga en estas tablas, normalmente es `1.0`, es decir, el rango de índices del vector coincide con el número de elementos que se espera sean insertados. A medida que aumenta el factor de carga, la eficiencia de la búsqueda de un elemento en la lista disminuye. Sin embargo no es conveniente que disminuya mucho el factor de carga, debido a que cuando éste es menor que `1.0` aumenta la memoria no utilizada.

> ***Nota***
>
> El principal inconveniente del direccionamiento con encadenamiento es el espacio adicional de cada elemento, necesario para enlazar un nodo de la lista con el siguiente.

12.7. REALIZACIÓN DE UNA TABLA DISPERSA ENCADENADA

El desarrollo de las operaciones de una tabla dispersa encadenada se basa en las operaciones del *TAD* `Lista`. Los pasos que se siguen son los siguientes:

1. Declarar los tipos de datos ajustándose a los elementos de la aplicación.
2. Definir la función *hash* con la que se van a obtener los índices de la tabla.
3. Inicializar la estructura como un *array* de listas vacías.

4. Definir la operación *insertar* un elemento. La inserción invoca al método que añade un nodo como primer elemento de la lista.
5. Definir la operación de *eliminar* un elemento. Esta operación busca el elemento dando su clave; antes de llamar al método que borra un nodo de una lista enlazada, se pide que el usuario confirme la acción de quitar el nodo.
6. Definir el método de búsqueda en la tabla de un elemento. Para realizar la operación, se pide la clave del elemento a buscar; si la búsqueda tiene éxito, devuelve la dirección del nodo, en caso contrario, `null`.

El desarrollo de las operaciones de la tabla dispersa se realiza en el contexto de una aplicación. Los elementos de la tabla son, a título de ejemplo, socios del club de montaña *La Parra*. Cada socio se identifica con estos datos: *número de socio, nombre completo, edad, sexo, fecha de alta*. Se elige como clave el número de socio, que es un entero en el rango de `101` a `1999`. Actualmente, el club tiene `94` socios, entonces la tabla dispersa se diseña con el tamaño de `97` posiciones (no hay perspectivas de mucho crecimiento). El factor de carga, con este supuesto, estará en valores próximos a uno.

La función *hash* utiliza el método de la multiplicación, genera valores dispersos entre `0` y `96`. La declaración de la clase `Elemento`:

```
public class TipoSocio
{
   class Fecha
   {
      int dia;
      int mes;
      int anno;
      Fecha (int d, int m, int a)
      {
         dia = d;
         mes = m;
         anno = a;
      }
      public String toString()
      {
         return " " + dia + "-" + mes + "-" + anno;
      }
   };

   int codigo;
   private String nombre;
   private int edad;
   Fecha f;
   public TipoSocio(String n, int c, int e, int d, int m, int a)
   {
      codigo = c;
      nombre = n;
      edad = e;
      f = new Fecha(d, m, a);
   }
   public int getCodigo()
   {
      return codigo;
   }
   // métodos de interfaz
```

```
};

public class Elemento
{
   TipoSocio socio;
   Elemento sgte;

   public Elemento(TipoSocio e)
   {
      socio = e;
      sgte = null;
   }
   public TipoSocio getSocio()
   {
      return socio;
   }
};
```

Función *hash,* método de multiplicación

En el apartado 12.2.4 se explica este método de generar índices dispersos aplicando el método de multiplicación. El rango de la dispersión depende del tamaño de la tabla, `M`. La clase declara la constante `R` y el método `dispersion()`:

```
static final double R = 0.618034;

static int dispersion(long x)
{
   double t;
   int v;
   t = R * x - Math.floor(R * x);   // parte decimal
   v = (int) (M * t);
   return v;
}
```

Inicializar la tabla dispersa

La operación inicial que se hace con la tabla, antes de dar de alta elementos, consiste en establecer cada posición del *array* a la condición de lista vacía; la forma de asegurar que esta operación se realiza es situándola en el constructor de `TablaDispersa`. La implementación consta de un bucle, con tantas iteraciones como tamaño de la tabla, en el que se asigna `null` a cada posición de la tabla.

```
public TablaDispersaEnlazada()  // constructor
{
   tabla = new Elemento[M];
   for (int k = 0; k < M; k++)
      tabla[k] = null;
   numElementos = 0;
}
```

12.7.1. Dar de alta un elemento

Para dar de alta un elemento, primero se determina el número de lista enlazada que le corresponde según el índice que devuelve la función de dispersión; a continuación, se inserta como primer nodo de la lista. El modo de inserción elegido es el más eficiente, ya que accede directamente a la posición de inserción, no necesita recorrer la lista, por ello la complejidad de la operación es `O(1)`.

La operación, en el contexto de almacenar socios, recibe un objeto *Socio,* crea un objeto `Elemento` (nodo de la lista) con el *Socio* y se inserta.

```
public void insertar(TipoSocio s)
{
   int posicion;
   Elemento nuevo;

   posicion = dispersion(s.getCodigo());
   nuevo = new Elemento(s);
   nuevo.sgte  = tabla[posicion];
   tabla[posicion] = nuevo;
   numElementos++;
}
```

12.7.2. Eliminar un elemento

La supresión de un elemento se hace dando como entrada la clave del elemento, que en el supuesto estudiado es el número de socio. Con la función de dispersión se obtiene el número de lista donde se encuentra, para hacer una búsqueda secuencial dentro de ella. Una vez encontrado el elemento, se muestran los campos y se pide *conformidad*. La retirada del nodo se hace enlazando el nodo anterior con el nodo siguiente, por lo que la búsqueda mantiene en la variable `anterior` la dirección del anterior al nodo actual.

El tiempo de ejecución de esta operación depende de la longitud de la lista enlazada en la que se busca el elemento; la longitud media de cada lista enlazada (número de nodos) es el factor de carga (véase apartado 12.6.2).

```
boolean conforme(TipoSocio a);

public void eliminar(int codigo)
{
   int posicion;

   posicion = dispersion(codigo);
   if (tabla[posicion] != null)  // no está vacía
   {
      Elemento anterior, actual;

      anterior = null;
      actual = tabla[posicion];

      while ((actual.sgte != null) &&
            actual.getSocio().getCodigo() != codigo)
      {
         anterior = actual;
         actual = actual.sgte;
      }
      if (actual.getSocio().getCodigo() != codigo)
         System.out.println("No se encuentra en la tabla el socio "
                                                          + codigo);
      else if (conforme (actual.getSocio()))   //se retira el nodo
      {
         if (anterior == null)                 // primer nodo
            tabla[posicion] = actual.sgte;
```

```
         else
            anterior.sgte = actual.sgte;

         actual = null;
         numElementos--;
      }
   }
}
```

12.7.3. Buscar un elemento

El algoritmo de búsqueda de un elemento es similar a la búsqueda que realiza la operación *eliminar*. El método `buscar()` devuelve la dirección del nodo (referencia) que contiene la clave de búsqueda, que en el contexto de la tabla de socios se corresponde con el código de socio. Si no se encuentra en la lista enlazada devuelve `null`.

```
public Elemento buscar(int codigo)
{
   Elemento p = null;
   int posicion;

   posicion = dispersion(codigo);

   if (tabla[posicion] != null)
   {
      p = tabla[posicion];
      while ((p.sgte != null) && p.socio.codigo != codigo)
         p = p.sgte;
      if (p.socio.codigo != codigo)
         p = null;
   }
   return p;
}
```

RESUMEN

Las tablas de dispersión son estructuras de datos para las cuales la complejidad de las operaciones básicas *insertar, eliminar* y *buscar* es constante. Cuando se utilizan tablas de dispersión los elementos que se guardan tienen que estar identificados por un campo clave, de tal forma que dos elementos distintos tengan claves distintas. La correspondencia entre un elemento y el índice de la tabla se hace con las funciones de transformación de claves, funciones *hash,* que convierten la clave de un elemento en un índice o posición de la tabla.

Dos criterios se deben seguir al elegir una *función hash,* `h(x)`. En primer lugar, que la complejidad de la función `h(x)` sea constante y fácil de evaluar. En segundo lugar, `h(x)` debe distribuir uniformemente las direcciones sobre el conjunto de índices posibles, de forma que se minimice el número de *colisiones*. El fenómeno de la *colisión* se produce si dadas dos claves diferentes `c1` y `c2`, la función *hash* devuelve la misma dirección, `h(c1) = h(c2)`. Aun siendo muy buena la distribución que realice `h(x)`, siempre existe la posibilidad de que colisionen dos claves, por ello el estudio del *direccionamiento disperso* se divide en dos partes: búsqueda de *funciones hash* y *resolución de colisiones*. Hay muchas funciones *hash,* la más popular es la *aritmética modular,* que aplica la *teoría de restos* para obtener valores dispersos dentro del rango determinado por el tamaño de la tabla, que debe elegirse como un número primo.

Se consideran dos modelos para resolver colisiones: *exploración* de direcciones y *hashing* enlazado. Las tablas *hash* con exploración resuelven las colisiones examinando una secuencia de posiciones de la tabla a partir de la posición inicial, determinada por `h(x)`. El método *exploración lineal* examina secuencialmente las claves hasta encontrar una posición vacía (caso de insertar); tiene el problema de la agrupación de elementos en posiciones contiguas, que afecta negativamente a la eficiencia de las operaciones. El método *exploración cuadrática* examina las posiciones de la tabla $p, p+1, p+2^2, ..., p+i^2$ considerando la tabla como un *array* circular. Para asegurar la eficiencia de la exploración cuadrática, el tamaño de la tabla debe elegirse como un número primo, y el *factor de carga* no debe superar el 50%.

Las tablas dispersas enlazadas se basan en utilizar listas (cadenas), de tal forma que en cada lista se colocan los elementos que tienen la misma dirección *hash*. Las operaciones *insertar, buscar* y *eliminar*, determinan primero el índice de la lista que le corresponde con la función *hash,* a continuación aplican la operación correspondiente del *TAD* `Lista`. Se aconseja que el factor de carga en las tablas enlazadas sea próximo a 1, ya que en el caso de crezca mucho puede empeorar la eficiencia de la búsqueda al aumentar el número de nodos de las listas.

Una de las aplicaciones de las tablas *hash* está en los compiladores de lenguajes de programación. La tabla donde se guardan los identificadores del programa, *tabla de símbolos,* se implementa como una tabla *hash.* La clave es el *símbolo* que se transforma en un valor entero para aplicar alguna de las funciones *hash*.

EJERCICIOS

12.1. Se tiene una aplicación en la que se espera que se manejen `50` elementos. ¿Cuál es el tamaño apropiado de una tabla *hash* que los almacene?

12.2. Una tabla *hash* se ha implementado con exploración lineal, considerando que actualmente el factor de carga es `0.30`. ¿Cuál es el número esperado de posiciones de la tabla que se prueban en una búsqueda de una clave, con éxito y sin éxito?

12.3. Para la secuencia de claves `29, 41, 22, 31, 50, 19, 42, 38`, una tabla *hash* de tamaño `12` y la función *hash aritmética modular,* mostrar las posiciones de almacenamiento suponiendo la exploración lineal.

12.4. Para la misma secuencia de claves del Ejercicio 12.3, igual tamaño de la tabla y la misma función *hash,* mostrar la secuencia de almacenamiento con la exploración cuadrática.

12.5. Se va a utilizar como función *hash* el método *mitad del cuadrado*. Si el tamaño de la tabla es de `100`, encontrar los índices que le corresponden a las claves `2134, 5231, 2212, 1011`.

12.6. Para la resolución de colisiones se utiliza el método de doble función *hash,* siendo la primera función el método *aritmética modular* y la segunda el método de la *multiplicación*. Encontrar las posiciones que ocupan los elementos con claves `14, 31, 62, 26, 39, 44, 45, 22, 15, 16` en una tabla de tamaño `17`.

12.7. Escribir el método `estaVacia()` para determinar que una tabla dispersa con exploración no tiene ningún elemento activo.

12.8. Escribir el método `estaVacia()` para determinar que una tabla dispersa enlazada no tiene elementos asignados.

12.9. Una tabla de dispersión con exploración tiene asignada n elementos. Demostrar que si el factor de carga es λ y la función *hash* distribuye uniformemente las claves, entonces se presentan `(n – 1)* λ/2` pruebas de posiciones de la tabla cuando se quiere insertar un elemento con una clave previamente insertada.

12.10. En el *hashing* enlazado, la operación de insertar un elemento se ha realizado como primer elemento de la lista. Si la inserción en la lista enlazada se hace de tal forma que esté ordenada respecto a la clave. ¿Qué ventajas y desventajas tiene respecto a la eficiencia de las operaciones *insertar, buscar* y *eliminar*?

12.11. Considérese esta otra estrategia para resolver colisiones en una tabla: en un vector auxiliar, *vector de desbordamiento*, se sitúan los elementos que colisionan con una posición ya ocupada, en orden relativo al campo clave. ¿Qué ventajas y desventajas pueden encontrarse en cuanto a la eficiencia de las operaciones de la tabla?

12.12. Empleando una tabla de dispersión con exploración cuadrática para almacenar 1000 cadenas de caracteres, el máximo valor que se quiere alcanzar del factor de carga es 0.6; suponiendo que las cadenas tienen un máximo de 12 caracteres, calcular:

a) El tamaño de la tabla.

b) La memoria total que necesita la tabla.

PROBLEMAS

12.1. Escribir el método `posicion()`, que tenga como entrada la clave de un elemento que representa los coches de alquiler de una compañía y que devuelva una dirección dispersa en el rango `0 .. 99`. Utilizar el método *mitad del cuadrado*, siendo la clave el número de matrícula, pero sólo considerando los caracteres de orden par.

12.2. Escribir el método `plegado()` para dispersar en el rango de `0` a `997`, considerando como clave el número de la Seguridad Social. Utilizar el método de plegamiento.

12.3. Se desea almacenar en un archivo los atletas participantes en un cross popular. Se ha establecido un máximo de participantes, `250`. Los datos de cada atleta son: *Nombre, Apellido, Edad, Sexo, Fecha de nacimiento y Categoría (Junior, Promesa, Senior, Veterano)*. Supóngase que el conjunto de direcciones del que se dispone es de `400`, en el rango de `0` a `399`. Se elige como campo clave el *Apellido* del atleta. La función *hash* a utilizar es la de *aritmética modular*. Antes de invocar a la función *hash* es necesario transformar la cadena a valor numérico (considerar los caracteres de orden impar, con un máximo de 5). Las colisiones se pueden resolver con el método de *exploración (prueba) lineal*.

Las operaciones que contempla el ejercicio son las de dar de alta un nuevo registro; modificar datos de un registro, eliminar un registro y buscar un atleta.

12.4. Las palabras de un archivo de texto se quieren mantener en una tabla *hash* con *exploración cuadrática* para poder hacer consultas rápidas. Los elementos de la tabla son la cadena con la palabra y el número de línea en el que aparece, sólo se consideran las 10 primeras ocurrencias de una palabra. Escribir un programa que lea el archivo según se vaya capturando una palabra e inserte el número de línea en la tabla dispersa. Téngase en cuenta que una palabra puede estar ya en la tabla, si es así se añade el número de línea.

12.5. Para comparar los resultados teóricos de la eficiencia de las tablas de dispersión, escribir un programa que inserte `1.000` enteros generados aleatoriamente en una tabla dispersa con *exploración lineal*. Cada vez que se inserte un nuevo elemento, contabilizar el número de posiciones de la tabla que se exploran. Calcular el número medio de posiciones exploradas para los factores de carga: `0.1, 0.2, ... 0.9`.

12.6. Escribir un programa para comparar los resultados teóricos de la eficiencia de las tablas de dispersión con *exploración cuadrática*. Seguir las pauta marcadas en el Problema 12.5.

12.7. Determinar la lista enlazada más larga (mayor número de nodos) cuando se utiliza una tabla *hash* enlazada para guardar `1.000` números enteros generados aleatoriamente. Escribir un programa que realice la tarea propuesta, considerando que el factor de carga es 1.

12.8. La realización de una tabla dispersa con exploración puede hacerse de tal manera que el tamaño de la tabla se establezca dinámicamente. Con estos vectores dinámicos se puede hacer una *reasignación* de tal forma que, si el factor de carga alcanza un determinado valor, se amplíe el tamaño en una cantidad determinada y se vuelvan a asignar los elementos, pero calculando un nuevo índice disperso, ya que el tamaño de la tabla ha aumentado. Se pide escribir un método que implemente este tipo de *reasignación*.

12.9. Para una tabla dispersa con exploración, escribir un método que determine el numero de elementos dados de baja.

12.10. Partiendo del método escrito en el Problema 12.9, añadir el código necesario para cuando el número de elementos eliminados supere el `10%` se produzca una *reasignación,* de tal forma que el factor de carga supere el `40%`.

12.11. En una tabla de dispersión enlazada, la eficiencia de la búsqueda disminuye según aumenta la longitud media de las listas enlazadas. Se quiere realizar una *reasignación* cuando la longitud media supere un factor determinado. La tabla dispersa, inicialmente, es de tamaño `M`, y la función de dispersión es aritmética modular: `h(x) = x modulo m`. Cada ampliación incrementa el tamaño de la tabla en 1, así en la primera el número de posiciones será `M+1`, en el rango de `0` a `M`; por esa razón se crea la lista en la posición `M`, y los elementos que se encuentran en la lista `0` se dispersan utilizando la función h_1`(x) = x modulo 2* M`. La segunda ampliación supone que el tamaño de la tabla sea `M +2`, en el rango de `0` a `M +1`, entonces se crea la lista de índice `M +1`, y los elementos que se encuentran en la lista `1` se dispersan utilizando la función h_2`(x) = x modulo 2*(M+1)`. Así sucesivamente, se va ampliando la estructura que almacena los elementos de una tabla dispersa enlazada.

Escribir los métodos que implementan la operación *insertar* siguiendo la estrategia indicada.

12.12. Escribir un programa que utilice la tabla *hash* enlazada descrita en el Problema 12.11 para guardar `1.000` números enteros generados aleatoriamente. Inicialmente, el tamaño de la tabla es `50`, cada vez que la longitud media de las cadenas (listas enlazadas) sea `4.5` se debe ampliar la tabla dispersa según el método descrito en el Problema 12.11.

CAPITULO 13

Árboles. Árboles binarios y árboles ordenados

Objetivos

Con el estudio de este capítulo, usted podrá:

- Estructurar datos en orden jerárquico.
- Conocer la terminología básica relativa a árboles.
- Distinguir los diferentes tipos de árboles binarios.
- Recorrer un árbol binario de tres formas diferentes.
- Reconocer la naturaleza recursiva de las operaciones con árboles.
- Representar un árbol binario con una estructura enlazada.
- Evaluar una expresión algebraica utilizando un árbol binario.
- Construir un árbol binario ordenado (de búsqueda).

Contenido

Conceptos clave

- ♦ Árbol.
- ♦ Árbol binario.
- ♦ Árbol binario de búsqueda.
- ♦ En orden.
- ♦ Hoja.
- ♦ Jerarquía.
- ♦ *Postorden*.
- ♦ *Preorden*.
- ♦ Raíz.
- ♦ Recorrido.
- ♦ Subárbol.

INTRODUCCIÓN

El árbol es una estructura de datos muy importante en informática y en ciencias de la computación. Los árboles son estructuras *no lineales,* al contrario que los *arrays* y las listas enlazadas, que constituyen *estructuras lineales.*

Los árboles se utilizan para representar fórmulas algebraicas, para organizar objetos en orden de tal forma que las búsquedas sean muy eficientes y en aplicaciones diversas tales como inteligencia artificial o algoritmos de cifrado. Casi todos los sistemas operativos almacenan sus archivos en árboles o estructuras similares a árboles. Además de las aplicaciones citadas, los árboles se utilizan en diseño de compiladores, procesado de texto y algoritmos de búsqueda.

En este capítulo se estudiarán el concepto de árbol general y los tipos de árboles más usuales, binario y binario de búsqueda o árbol ordenado. También se estudiarán algunas aplicaciones típicas del diseño y la construcción de árboles.

13.1. ÁRBOLES GENERALES Y TERMINOLOGÍA

Intuitivamente, el concepto de árbol implica una estructura en la que los datos se organizan de modo que los elementos de información están relacionados entre sí a través de ramas. El árbol genealógico es el ejemplo típico más representativo del concepto de árbol general. La Figura 13.1 representa un ejemplo de árbol general, gráficamente puede verse como un árbol invertido, con la raíz en la parte más alta, de la que salen ramas que llegan a las hojas, que están en la parte baja.

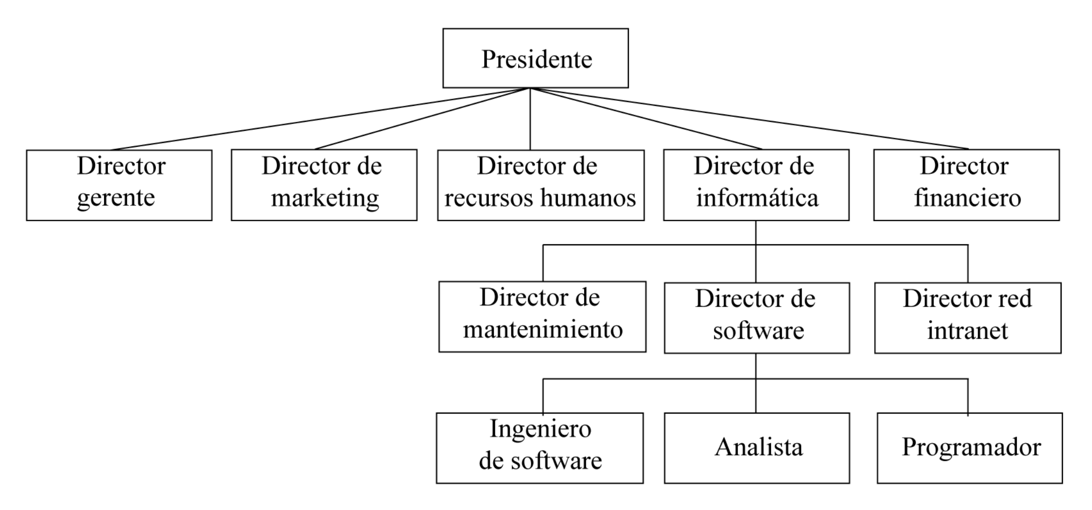


Figura 13.1 Estructura jerárquica tipo árbol

Un **árbol** consta de un conjunto finito de elementos, denominados **nodos** y de un conjunto finito de líneas dirigidas, denominadas **ramas,** que conectan los nodos. El número de ramas asociado con un nodo es el **grado** del nodo.

> ***Definición 1***
>
> Un árbol consta de un conjunto finito de elementos, llamados nodos y de un conjunto finito de líneas dirigidas, llamadas ramas, que conectan los nodos.

Definición 2

Un árbol es un conjunto de uno o más nodos tales que:

1. Hay un nodo diseñado especialmente llamado raíz.
2. Los nodos restantes se dividen en `n ≥ 0` conjuntos disjuntos, `T1 ... Tn`, tal que cada uno de estos conjuntos es un árbol. A `T1 ... Tn` se les denomina subárboles del raíz.

Si un árbol no está vacío, entonces el primer nodo se llama **raíz.** Obsérvese en la definición 2 que el árbol ha sido definido de modo recursivo, ya que los subárboles se definen como árboles. La Figura 13.2 muestra un árbol.

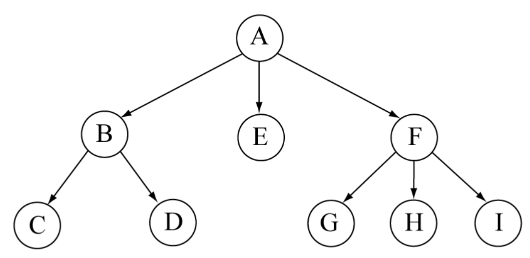


Figura 13.2 Árbol

13.1.1. Terminología

Además del nodo raíz, existen muchos términos utilizados en la descripción de los atributos de un árbol. En la Figura 13.3, el nodo `A` es el raíz. Utilizando el concepto de árboles genealógicos, un nodo puede ser considerado como **padre** si tiene nodos sucesores.

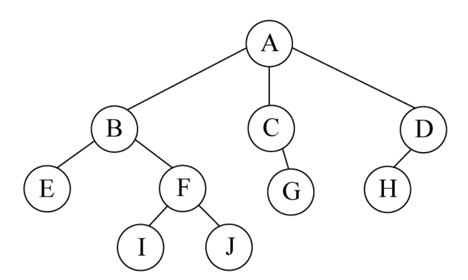


Figura 13.3 Árbol general

Estos nodos sucesores se llaman **hijos.** Por ejemplo, el nodo `B` es el padre de los hijos `E` y `F`. El padre de `H` es el nodo `D`. Un árbol puede representar diversas generaciones en la familia. Los hijos de un nodo y los hijos de estos hijos se llaman **descendientes,** y el padre y los abuelos de un nodo son sus **ascendientes.** Por ejemplo, los nodos `E, F, I` y `J` son descendientes de `B`. Cada nodo no raíz tiene un único padre y cada padre tiene cero o más nodos hijos. Dos o más nodos con el mismo padre se llaman **hermanos.** Un nodo sin hijos, tal como `E, I, J, G` y `H` se llama nodo **hoja**.

El **nivel** de un nodo es su distancia al nodo raíz. La raíz tiene una distancia cero de sí misma, por ello se dice que está en el nivel 0. Los hijos del nodo raíz están en el nivel 1, sus hijos están en el nivel 2, y así sucesivamente. Una cosa importante que se aprecia entre los niveles de nodos es la

relación entre niveles y *hermanos*. Los hermanos están siempre al mismo nivel, pero no todos los nodos de un mismo nivel son necesariamente hermanos. Por ejemplo, en el nivel 2 (Figura 13.4), `C` y `D` son hermanos, al igual que lo son `G`, `H`, e `I`, pero `D` y `G` no son hermanos, ya que ellos tienen diferentes padres.

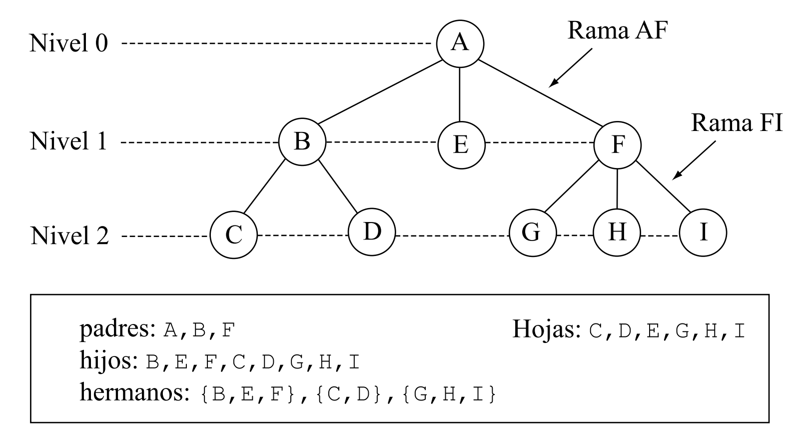


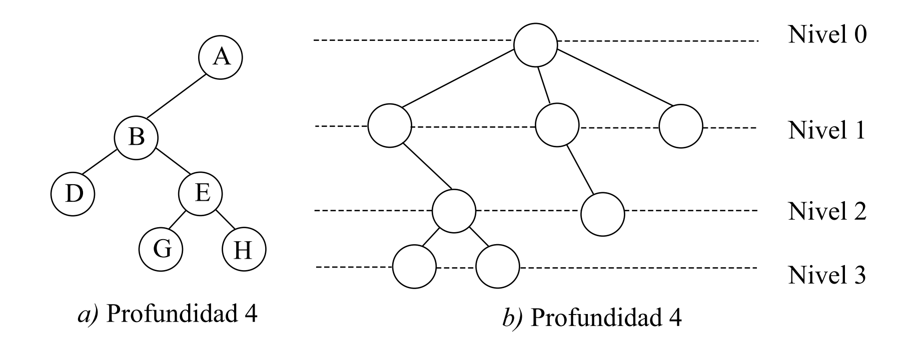
Figura 13.4 Terminología de árboles

Un **camino** es una secuencia de nodos en los que cada nodo es adyacente al siguiente. Cada nodo del árbol puede ser alcanzado (se llega a él) siguiendo un único camino que comienza en el nodo raíz. En la Figura 13.4, el camino desde el raíz a la hoja `I`, se representa por `AFI`. Incluye dos ramas distintas `AF` y `FI`.

La **altura** o **profundidad** de un árbol es el nivel de la hoja del camino más largo desde la raíz más uno. Por definición[1], la altura de un árbol vacío es `0`. La Figura 13.4 contiene nodos en tres niveles: `0`, `1` y `2`. Su altura es `3`.

Definición

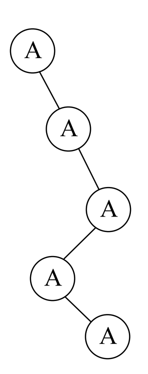
El nivel de un nodo es su distancia desde el nodo raíz. La altura de un árbol es el nivel de la hoja del camino más largo desde el raíz más uno.



a) Profundidad 4 *b)* Profundidad 4

---

[1] También se suele definir la **profundidad** de un árbol como el nivel máximo de cada nodo. En consecuencia, la profundidad del nodo raíz es 0, la de su hijo 1, etc. Las dos terminologías son aceptadas.



c) Profundidad 5

Figura 13.5 Árboles de profundidades diferentes

Un árbol se divide en subárboles. Un **subárbol** es cualquier estructura conectada por debajo del nodo raíz. Cada nodo de un árbol es la raíz de un subárbol que se define por el nodo y todos sus descendientes. El primer nodo de un subárbol se conoce como el nodo raíz del subárbol y se utiliza para nombrar el subárbol. Además, los subárboles se pueden subdividir en subárboles. En la Figura 13.4, `BCD` es un subárbol al igual que `E` y `FGHI`. Obsérvese que, por esta definición, un nodo simple es un subárbol. Por consiguiente, el subárbol `B` se puede dividir en subárboles `C` y `D` mientras que el subárbol `F` contiene los subárboles `G, H` e `I`. Se dice que `G, H, I, C` y `D` son subárboles sin descendientes.

Definición recursiva

El concepto de subárbol conduce a una definición recursiva de un árbol. Un árbol es un conjunto de nodos que:

1. Es vacío.
2. O tiene un nodo determinado, llamado raíz, del que jerárquicamente descienden cero o más subárboles, que son también árboles.

Resumen de definiciones

- El primer nodo de un árbol, normalmente dibujado en la posición superior, se denomina **raíz** del árbol.
- Las flechas que conectan un nodo con otro se llaman **arcos** o **ramas.**
- Los **nodos terminales,** esto es, nodos de los cuales no se deduce ningún nodo, se denominan **hojas.**
- Los nodos que no son hojas se denominan **nodos internos.**
- En un árbol donde una rama va de un nodo n_1 a un nodo n_2, se dice que n_1 es el padre de n_2 y que n_2 es un **hijo** de n_1.
- n_1 se llama **ascendiente** de n_2 si n_1 es el padre de n_2 o si n_1 es el padre de un ascendiente de n_2.
- n_2 se llama **descendiente** de n_1 si n_1 es un ascendiente de n_2.

- Un **camino** de n_1 a n_2 es una secuencia de arcos contiguos que van de n_1 a n_2.
- La **longitud** de un camino es el número de arcos que contiene o, de forma equivalente, el número de nodos del camino menos uno.
- El **nivel** de un nodo es la longitud del camino que lo conecta al nodo raíz.
- La **profundidad** o **altura** de un árbol es la longitud del camino más largo que conecta el raíz a una hoja.
- Un **subárbol** de un árbol es un subconjunto de nodos del árbol, conectados por ramas del propio árbol, esto es, a su vez un árbol.
- Sea `S` un subárbol de un árbol `A`: si para cada nodo *n* de `SA`, `SA` contiene también todos los descendientes de *n* en `A`, `SA` se llama un **subárbol completo** de `A`.
- Un árbol está **equilibrado** cuando, dado un número máximo *k* de hijos de cada nodo y la **altura del árbol** *h,* cada nodo de nivel $k < h\text{-}1$ tiene exactamente *k* hijos. El árbol está equilibrado perfectamente, si cada nodo de nivel $l<h$ tiene exactamente *k* hijos.

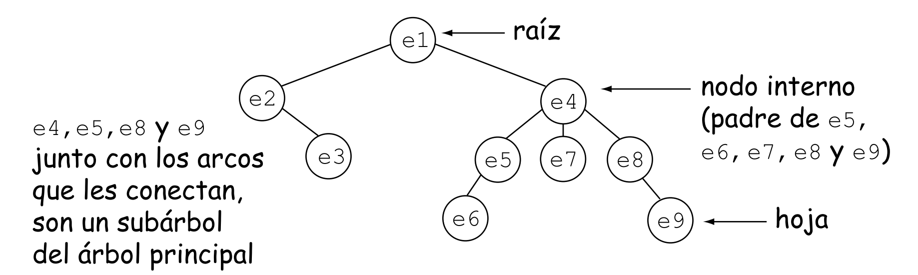


Figura 13.6 Un árbol y sus nodos

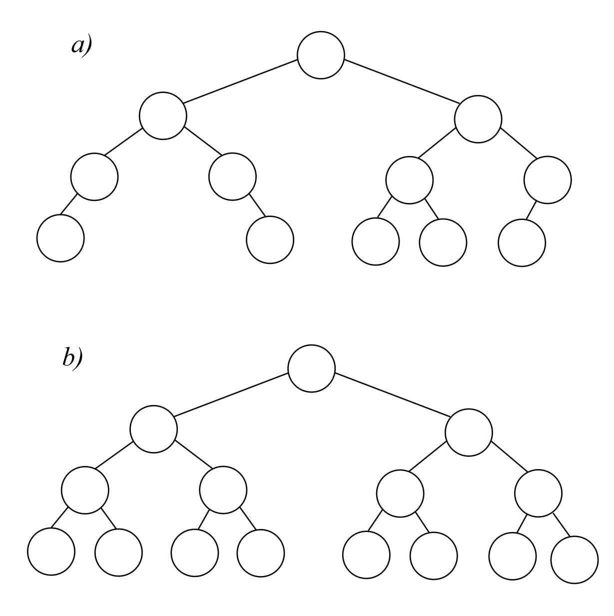


Figura 13.7 *a)* Un árbol equilibrado; *b)* un árbol perfectamente equilibrado

13.1.2. Representación gráfica de un árbol

Las formas más frecuentes de representar en papel un árbol son como árbol invertido y como una lista.

Representación como árbol invertido

Es el diagrama o carta de organización utilizado hasta ahora en las diferentes figuras. El nodo raíz se encuentra en la parte más alta de una jerarquía, de la que descienden ramas que van a parar a los nodos hijos, y así sucesivamente. La Figura 13.8 presenta esta representación para una descomposición de una computadora.

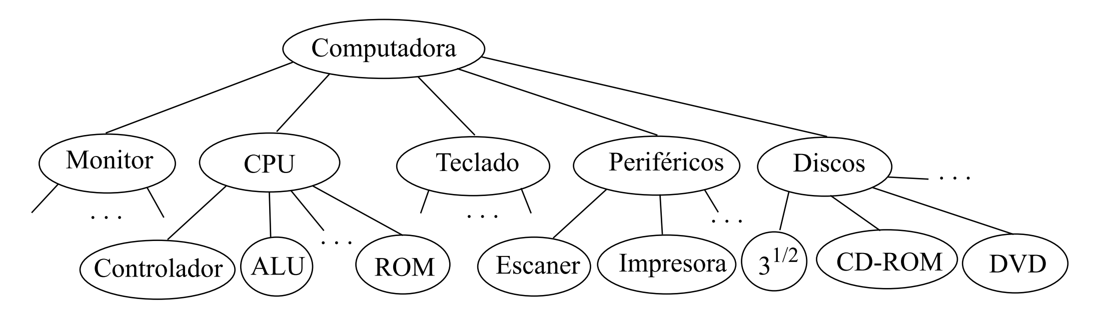


Figura 13.8 Árbol general (computadora)

Representación de lista

Otro formato utilizado para representar un árboles es la lista entre paréntesis. Esta es la notación utilizada con expresiones algebraicas. En esta representación, cada paréntesis abierto indica el comienzo de un nuevo nivel y cada paréntesis cerrado completa un nivel y se mueve hacia arriba un nivel en el árbol. La notación en paréntesis correspondiente al árbol de la Figura 13.2: `A(B (C, D), E, F, (G, H, I))`.

Ejemplo 13.1

Representar el árbol general de la Figura 13.9 en forma de lista.

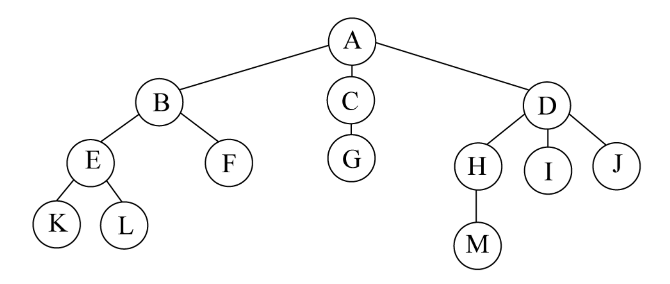


Figura 13.9 Árbol general

La solución es: `A (B (E (K, L), F), C (G), D (H (M), I, J)))`.

13.2. ÁRBOLES BINARIOS

Un **árbol binario** es un árbol cuyos nodos no pueden tener más de dos subárboles. En un árbol binario, cada nodo puede tener cero, uno o dos hijos (subárboles). Se conoce el nodo de la izquierda como *hijo izquierdo* y el nodo de la derecha como *hijo derecho*.

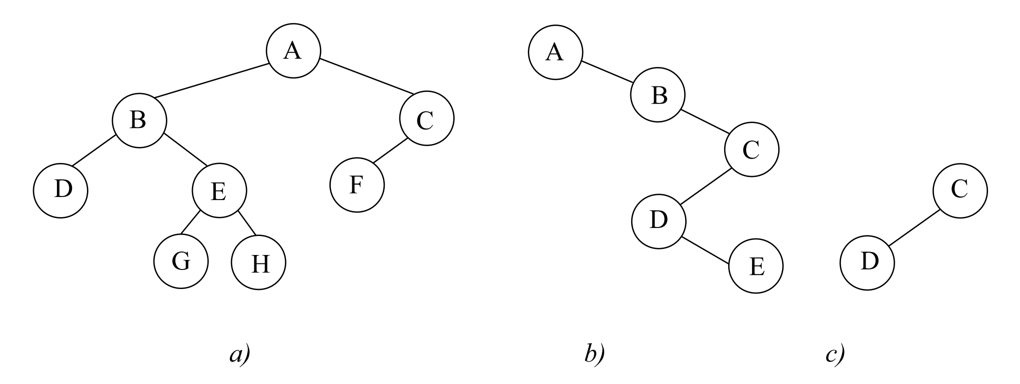


Figura 13.10 Árboles binarios

> ***Nota***
>
> Un árbol binario no puede tener más de dos subárboles.

Un árbol binario es una estructura recursiva. Cada nodo es la raíz de su propio subárbol y tiene hijos, que son raíces de árboles, llamados subárboles derecho e izquierdo del nodo, respectivamente. Un árbol binario se divide en tres subconjuntos disjuntos:

`{R}`	Nodo raíz.
`{I1, I2, ...In}`	Subárbol izquierdo de `R`.
`{D1, D2, ...Dn}`	Subárbol derecho de `R`.

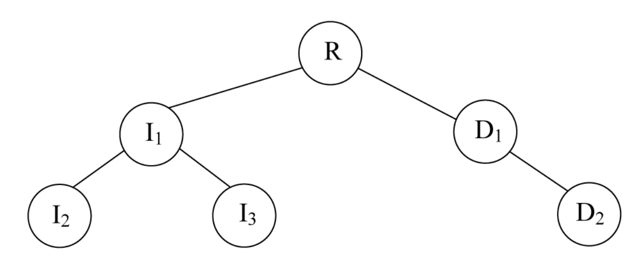


Figura 13.11 Árbol binario con subárbol izquierdo

En cualquier nivel *n,* un árbol binario puede contener de `1` a 2^n nodos. El número de nodos por nivel contribuye a la densidad del árbol.

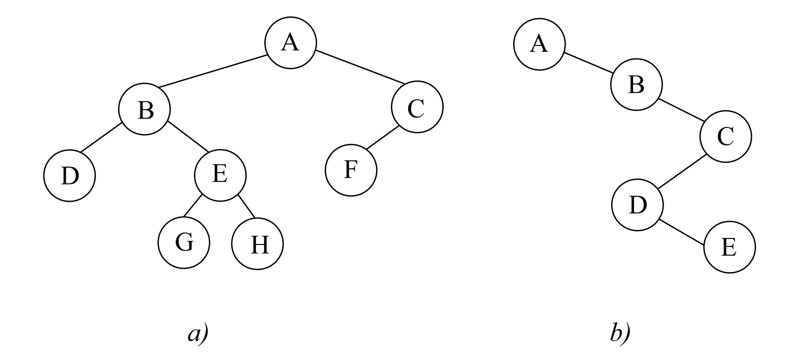


Figura 13.12 Árboles binarios: a) profundidad 4; b) profundidad 5

En la Figura 13.12*a*, el árbol `A` contiene `8` nodos en una profundidad de `4`, mientras que el árbol de la Figura 13.12*b* contiene `5` nodos y una profundidad `5`.

13.2.1. Equilibrio

La distancia de un nodo a la raíz determina la eficiencia con la que puede ser localizado. Por ejemplo, dado cualquier nodo de un árbol, a sus hijos se puede acceder siguiendo sólo un camino de bifurcación o de ramas, el que conduce al nodo deseado. De modo similar, a los nodos en el nivel 2 de un árbol sólo puede accederse siguiendo dos ramas del árbol.

La característica anterior nos conduce a una característica muy importante de un árbol binario, su **balance** o **equilibrio.** Para determinar si un árbol está equilibrado, se calcula su factor de equilibrio. El **factor de equilibrio** de un árbol binario es la diferencia en altura entre los subárboles derecho e izquierdo. Si la altura del subárbol izquierdo es h_I y la altura del subárbol derecho h_D, entonces el factor de equilibrio del árbol `B` se determina por la siguiente fórmula: $B = h_D - h_I$.

Utilizando esta fórmula, el equilibrio del nodo raíz los árboles de la Figura 13.12 son (a) `-1` y (b) `4`.

Un árbol está **perfectamente equilibrado** si su equilibrio o balance es *cero* y sus subárboles son también perfectamente equilibrados. Dado que esta definición ocurre raramente se aplica una definición alternativa: un árbol binario está equilibrado si la altura de sus subárboles difiere en no más de uno y sus subárboles son también equilibrados; por consiguiente, el factor de equilibrio de cada nodo puede tomar los valores `-1, 0, +1`.

13.2.2. Árboles binarios completos

Un árbol binario **completo** de profundidad *n* es un árbol en el que para cada nivel, del *0* al nivel *n-1,* tiene un conjunto lleno de nodos, y todos los nodos hoja a nivel *n* ocupan las posiciones más a la izquierda del árbol.

Un árbol binario completo que contiene 2^n nodos a nivel *n* es un **árbol lleno.** Un árbol lleno es un árbol binario que tiene el máximo número de entradas para su altura. Esto sucede cuando el último nivel está lleno. La Figura 13.13 muestra un árbol binario completo; el árbol de la Figura 13.14b se corresponde con uno lleno.

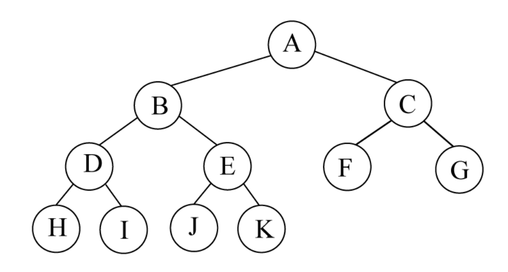


Figura 13.13 Árbol completo (Profundidad 4)

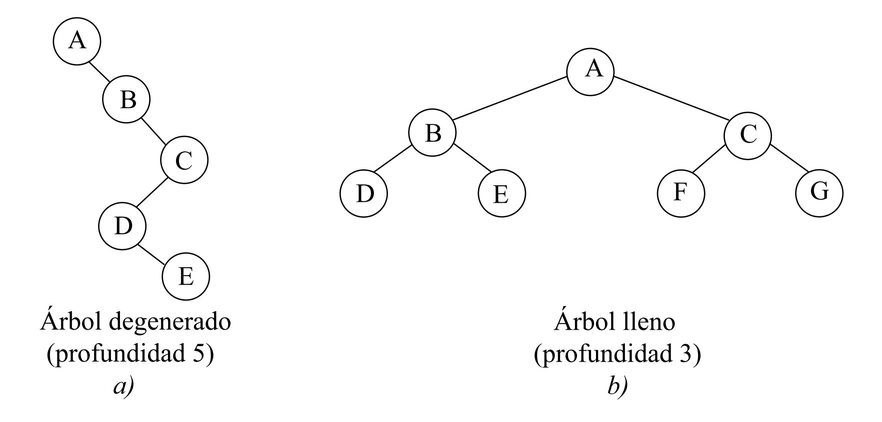


Figura 13.14 Clasificación de árboles binarios: a) degenerado; b) lleno

El último caso de árbol es un tipo especial, denominado **árbol degenerado,** en el que hay un solo nodo hoja (E) y cada nodo no hoja sólo tiene un hijo. Un árbol degenerado es equivalente a una lista enlazada.

Los árboles binarios completos y llenos de profundidad k+1 proporcionan algunos datos matemáticos de interés. En cada caso, existe un nodo (2^0) al nivel 0 (raíz), dos nodos (2^1) a nivel 1, cuatro nodos (2^2) a nivel 2, etc. A través de los primeros $k-1$ niveles se puede demostrar, considerando la suma de los términos de una progresión geométrica de razón 2, que hay $2^k - 1$ nodos.

$$1 + 2 + 4 + \ldots + 2^{k-1} = 2^k - 1$$

A nivel k, el número de nodos adicionados para un árbol completo está en el rango de un mínimo de 1 a un máximo de 2^k (lleno). Con un árbol lleno, el número de nodos es:

$$1 + 2 + 4 + \ldots + 2^{k-1} + 2^k = 2^{k+1} - 1$$

El número de nodos, n, en un árbol binario completo de profundidad k+1 (0 a k niveles) cumple la inigualdad:

$$2^k \leq n \leq 2^{k+1} - 1 < 2^{k+1}$$

Aplicando logaritmos a la ecuación con desigualdad anterior:

$$k \leq \log_2 (n) < k + 1$$

se deduce que la altura o profundidad de un árbol binario completo de `n` nodos es:

$h = |Log_2 n| + 1$ (parte entera de $Log_2 n + 1$)

Por ejemplo, un árbol lleno de profundidad 4 (niveles 0 a 3) tiene $2^4 - 1 = 15$ nodos.

Ejemplo 13.2

Calcular la profundidad máxima y mínima de un árbol con 5 nodos.

La profundidad máxima de un árbol con 5 nodos es 5, se corresponde con un árbol degenerado.

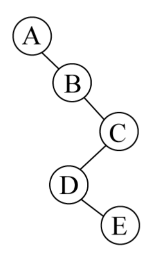


Figura 13.15 Árbol degenerado de raíz A

La profundidad mínima *h* (número de niveles más uno) de un árbol con 5 nodos, aplicando la inecuación del número de nodos de un árbol binario completo es:

$k \leq \log_2 (5) < k + 1$

como $\log_2(5) = 2.32$, la profundidad $h = 3$.

Ejemplo 13.3

Suponiendo que se tiene n = 10.000 elementos que van a ser los nodos de un árbol binario completo. Determinar la profundidad del árbol.

En el árbol binario completo con `n` nodos, la profundidad del árbol es el valor entero de $\log_2 n + 1$, que es a su vez la distancia del camino más largo desde la raíz a un nodo más uno.

$\text{Profundidad} = \text{int} (\log_2 10000) + 1 = \text{int} (13.28) + 1 = 14$

13.2.3. *TAD* Árbol binario

La estructura de árbol binario constituye un *tipo abstracto de datos;* las operaciones básicas que definen el *TAD árbol binario* son las siguientes:

Tipo de dato	Dato que se almacena en los nodos del árbol.
Operaciones	
CrearArbol	Inicia el árbol como vacío.
Construir	Crea un árbol con un elemento raíz y dos ramas, izquierda y derecha que son a su vez árboles.

EsVacio	Comprueba si el árbol no tiene nodos.
Raiz	Devuelve el nodo raíz.
Izquierdo	Obtiene la rama o subárbol izquierdo de un árbol dado.
Derecho	Obtiene la rama o subárbol derecho de un árbol dado.
Borrar	Elimina del árbol el nodo con un elemento determinado.
Pertenece	Determina si un elemento se encuentra en el árbol.

13.2.4. Operaciones en árboles binarios

Algunas de las operaciones típicas que se realizan en árboles binarios son las siguientes:

- Determinar su altura.
- Determinar su número de elementos.
- Hacer una copia.
- Visualizar el árbol binario en pantalla o en impresora.
- Determinar si dos árboles binarios son idénticos.
- Borrar (eliminar el árbol).
- Si es un árbol de expresión, evaluar la expresión.

Todas estas operaciones se pueden realizar recorriendo el árbol binario de un modo sistemático. El recorrido es la operación de visita al árbol o, lo que es lo mismo, la visita a cada nodo del árbol una vez y sólo una. La visita de un árbol es necesaria en muchas ocasiones; por ejemplo, si se desea imprimir la información contenida en cada nodo. Existen diferentes formas de visitar o recorrer un árbol que se estudiarán más adelante.

13.3. ESTRUCTURA DE UN ÁRBOL BINARIO

Un árbol binario se construye con nodos. Cada nodo debe contener el campo *dato* (datos a almacenar) y dos campos de enlace *(apuntador),* uno al subárbol izquierdo **(izquierdo, izdo)** y otro al subárbol derecho **(derecho, dcho)**. El valor `null` indica un árbol o un subárbol vacío.

La Figura 13.16 muestra la representación enlazada de dos árboles binarios de raíz `A`. El primero es un árbol degenerado a la izquierda; el segundo es un árbol binario completo de profundidad `4`.

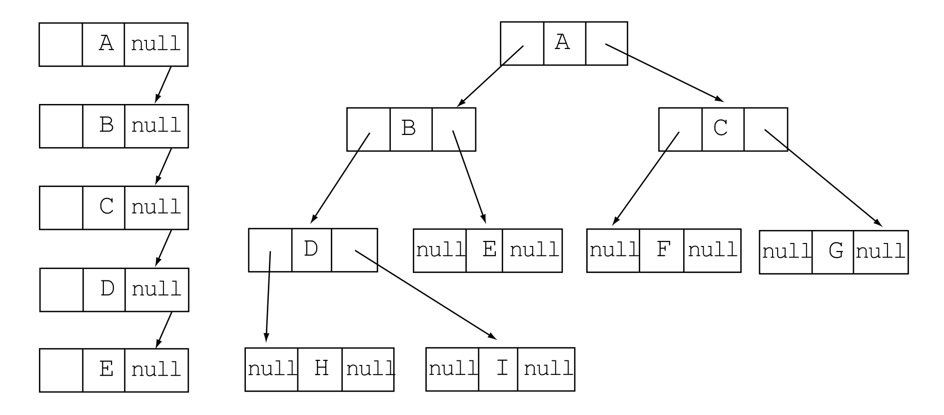


Figura 13.16 Representación enlazada de dos árboles binarios

Se puede observar que los nodos de un árbol binario que son hojas se caracterizan por tener sus dos campos de enlace a `null`.

13.3.1. Representación de un nodo

La clase `Nodo` agrupa a todos los campos de que consta: `dato`, `izdo` (rama izquierda) y `dcho` (rama derecha). Además, dispone de dos constructores; el primero inicializa el campo `dato` a un valor y los enlaces a `null`, en definitiva, se inicializa como hoja y el segundo inicializa `dato` a un valor y las ramas a dos subárboles.

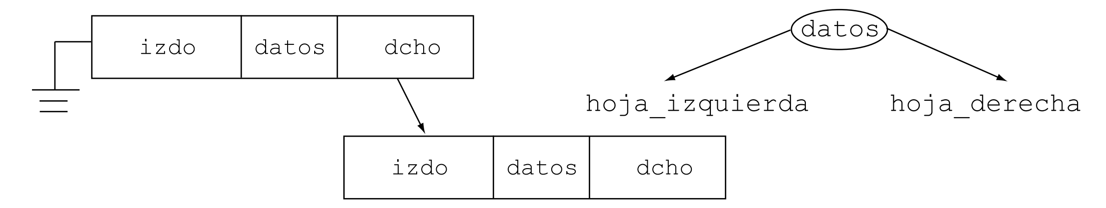


Figura 13.17 Representación gráfica de los campos de un nodo

```
package arbolBinario;
public class Nodo
{
   protected Object dato;
   protected Nodo izdo;
   protected Nodo dcho;
   public Nodo(Object valor)
   {
      dato = valor;
      izdo = dcho = null;
   }
   public Nodo(Nodo ramaIzdo, Object valor, Nodo ramaDcho)
   {
      this(dato);
      izdo = ramaIzdo;
      dcho = ramaDcho;
   }
   //    operaciones de acceso
   public Object valorNodo(){ return valor; }
   public Nodo subarbolIzdo(){ return izdo; }
   public Nodo subarbolDcho(){ return dcho; }
   public void nuevoValor(Object d){ dato = d; }
   public void ramaIzdo(Nodo n){ izdo = n; }
   public void ramaDcho(Nodo n){ dcho = n; }

}
```

13.3.2. Creación de un árbol binario

A partir del nodo raíz de un árbol se puede acceder a los demás nodos del árbol, por ello se mantiene la referencia a la raíz del árbol. Las ramas izquierda y derecha son, a su vez, árboles binarios que tienen su raíz, y así recursivamente hasta llegar a las hojas del árbol. La clase `ArbolBinario` tiene el campo `raiz`, un constructor que inicializa `raiz` y métodos para implementar las operaciones.

```
package arbolBinario;
public class ArbolBinario
{
   protected Nodo raiz;
   public ArbolBinario()
   {
   raiz = null;
   }
   public ArbolBinario(Nodo raiz)
   {
      this.raiz = raiz;
   }
   public Nodo raizArbol()
   {
   return raiz;
   }
   // Comprueba el estatus del árbol
   boolean esVacio()
   {
   return raiz == null;
   }
   // ...
```

El método `nuevoArbol()` crea un árbol de raíz un nodo con el campo dato, rama izquierda y derecha pasadas en los argumentos.

```
public static Nodo nuevoArbol(
                    Nodo ramaIzqda, Object dato, Nodo ramaDrcha)
{
     return new Nodo(ramaIzqda, dato, ramaDrcha);
}
```

Así, para crear el árbol binario de la Figura 13.18, se utiliza un esquema secuencial y con una `Pila` (véase el Capítulo 9) que guarda, en cada paso, los subárboles:

```
import TipoPila.PilaVector;

ArbolBinario arbol;
Nodo a1, a2, a;
PilaVector pila = new PilaVector();

a1 = ArbolBinario.nuevoArbol(null,"Maria",null);
a2 = ArbolBinario.nuevoArbol(null,"Rodrigo",null);
a = ArbolBinario.nuevoArbol(a1,"Esperanza",a2);
pila.insertar(a);

a1 = ArbolBinario.nuevoArbol(null,"Anyora",null);
a2 = ArbolBinario.nuevoArbol(null,"Abel",null);
a = ArbolBinario.nuevoArbol(a1,"M Jesus",a2);
pila.insertar(a);

a2 = (Nodo) pila.quitar();
a1 = (Nodo) pila.quitar();
a = ArbolBinario.nuevoArbol(a1,"Esperanza",a2);
arbol = new ArbolBinario(a);
```

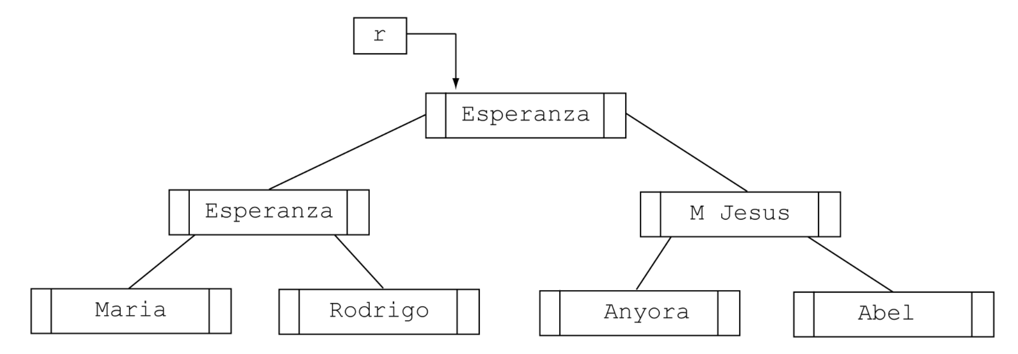


Figura 13.18 Árbol binario de cadenas

13.4. ÁRBOL DE EXPRESIÓN

Una aplicación muy importante de los árboles binarios son los *árboles de expresiones*. Una **expresión** es una secuencia de *tokens* (componentes de léxicos que siguen unas reglas establecidas). Un *token* puede ser un operando o bien un operador.

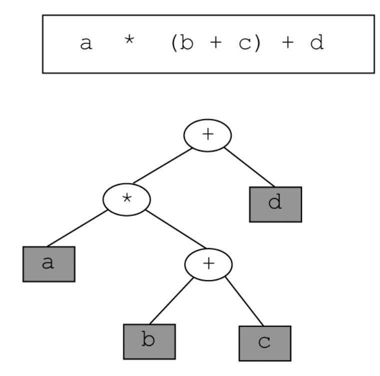


Figura 13.19 Una expresión infija y su árbol de expresión

La Figura 13.19 representa la expresión *infija* `a * (b + c) + d` junto a su árbol de expresión. El nombre de *infija* es debido a que los operadores se sitúan *entre* los operandos.

Un **árbol de expresión** es un árbol binario con las siguientes propiedades:

1. Cada hoja es un operando.
2. Los nodos raíz y los nodos internos son operadores.
3. Los subárboles son subexpresiones cuyo nodo raíz es un operador.

Los árboles binarios se utilizan para representar expresiones en memoria, esencialmente en compiladores de lenguajes de programación. Se observa que los paréntesis de la expresión no aparecen en el árbol, pero están implicados en su forma, y esto resulta muy interesante para la evaluación de la expresión.

Si se supone que todos los operadores tienen dos operandos, se puede representar una expresión mediante un árbol binario cuya raíz contiene un operador y cuyos subárboles izquierdo y derecho

son los operandos izquierdo y derecho, respectivamente. Cada operando puede ser una letra (`x, y, a, b` etc.) o una subexpresión representada como un subárbol. La Figura 13.20 muestra un árbol cuya raíz es el operador `*`, su subárbol izquierdo representa la subexpresión `(x + y)` y su subárbol derecho representa la subexpresión `(a - b)`. El nodo raíz del subárbol izquierdo contiene el operador `(+)` de la subexpresión izquierda y el nodo raíz del subárbol derecho contiene el operador `(-)` de la subexpresión derecha. Todos los operandos letras se almacenan en nodos hojas.

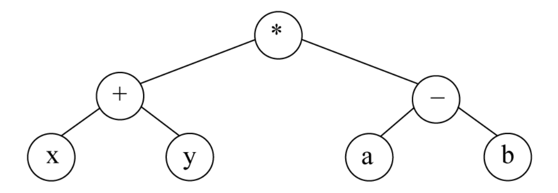


Figura 13.20 Árbol de expresión `(x + y)*(a - b)`

Utilizando el razonamiento anterior, la expresión `(x * (y - z)) + (a - b)` con paréntesis alrededor de las subexpresiones, forma el árbol binario de la Figura 13.21.

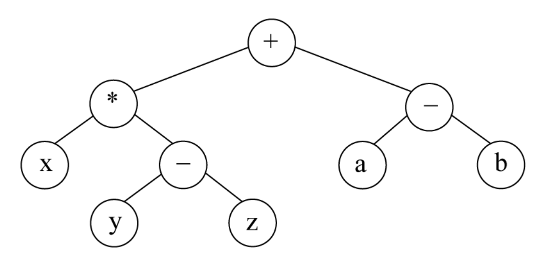


Figura 13.21 Árbol de expresión `(x * (y - z))+(a - b)`

Ejemplo 13.4

Deducir las expresiones que representan los árboles binarios de la Figura 13.22.

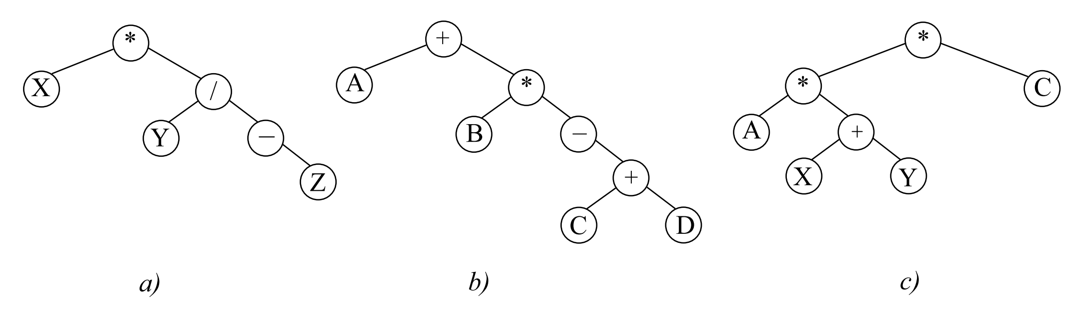


Figura 13.22 Diferentes árboles de expresión

Soluciones

a) `X * Y /(-Z)`

b) `A + B * (-(C + D))`

c) `A * (X + Y)) * C`

Ejemplo 13.5

Dibujar la representación en árbol binario de cada una de las siguientes expresiones:

a) `Y * X /(A + B)* C`

b) `X * Y /A + B * C`

Soluciones

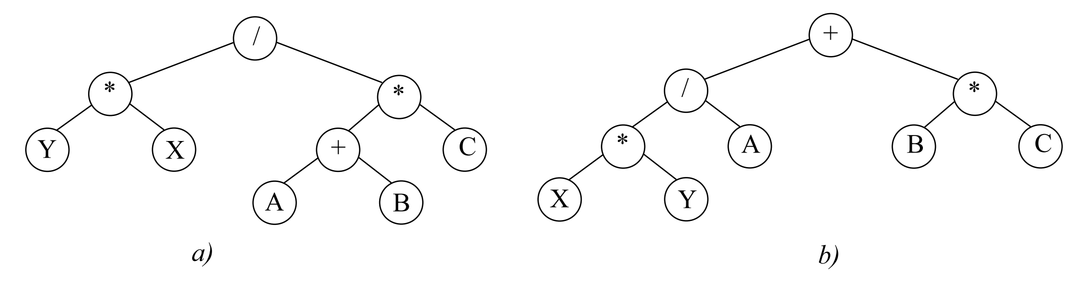


Figura 13.23 Árboles de expresión *a* y *b*

13.4.1. Reglas para la construcción de árboles de expresiones

Los árboles de expresiones se utilizan en las computadoras para evaluar expresiones usadas en programas. El algoritmo más sencillo para construir un árbol de expresión es aquel que lee una expresión completa que contiene paréntesis. Una expresión con paréntesis es aquella en que:

1. La prioridad se determina sólo por paréntesis.
2. La expresión completa se sitúa entre paréntesis.

A fin de ver la prioridad en las expresiones, considérese la expresión

```
a * c + e / g – (b + d)
```

Los operadores con prioridad más alta son `*` y /, es decir:

```
(a * c) + (e / g) – (b + d)
```

Los operadores que siguen en orden de prioridad son + y –, que se evalúan de izquierda a derecha. Por consiguiente, se puede escribir:

```
((a * c) + (e / g)) – (b + d)
```

Por último, la expresión completa entre paréntesis será:

```
(((a * c) + (e / g)) – (b + d))
```

El algoritmo para la construcción de un árbol de expresión puede expresarse en los siguientes pasos:

1. La primera vez que se encuentre un paréntesis a izquierda, crear un nodo que será el nodo raíz. Se llama a éste *nodo actual* y se sitúa en una pila.
2. Cada vez que se encuentre un nuevo paréntesis a izquierda, crear un nuevo nodo. Si el *nodo actual* no tiene un hijo izquierdo, hacer el nuevo nodo el hijo izquierdo; en caso contrario, hacerlo el hijo derecho. Hacer el nuevo nodo el *nodo actual* y ponerlo en la pila.

3. Cuando se encuentre un operando, crear un nuevo nodo y asignar el operando a su campo de datos. Si el *nodo actual* no tiene un hijo izquierdo, hacer el nuevo nodo el hijo izquierdo; en caso contrario, hacerlo el hijo derecho.
4. Cuando se encuentre un operador, sacar el nodo cabeza de la pila y situar el operador en el campo dato del nodo.
5. Ignorar el paréntesis derecho y los blancos.

Ejemplo 13.6

Determinar las expresiones correspondientes de los árboles de expresión de la Figura13.24

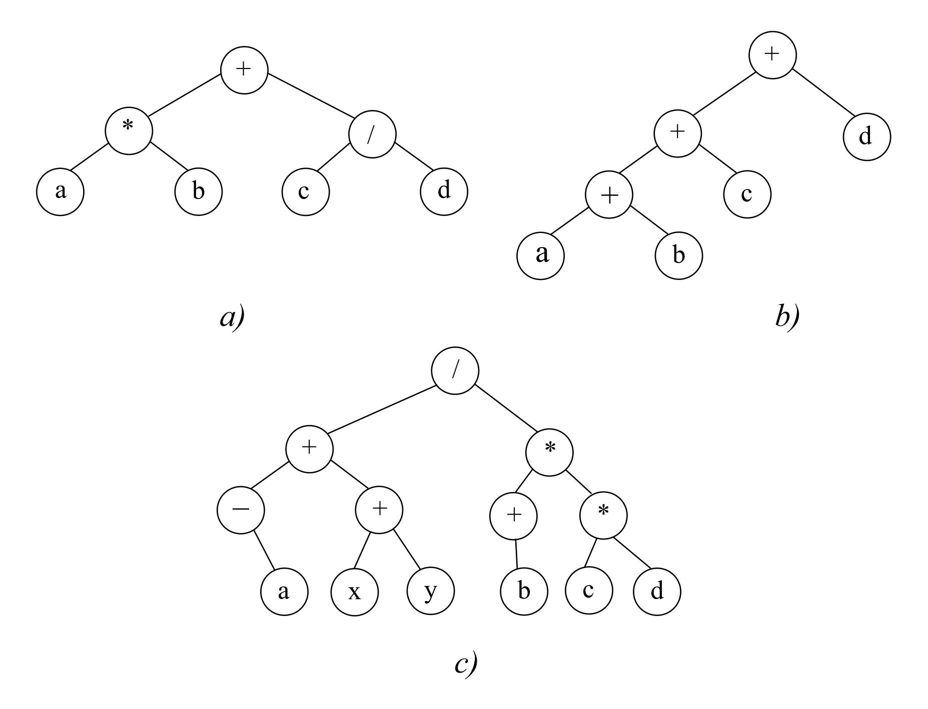


Figura 13.24 Árboles de expresión

Las soluciones correspondientes son:

a. `((a * b) + ( c / d))`

b. `(((a + b) + c) + d)`

c. `((( -a ) + (x + y)) / ((+b) * (c * d)))`

13.5. RECORRIDO DE UN ÁRBOL

Para visualizar o consultar los datos almacenados en un árbol se necesita *recorrer* el árbol o *visitar* los nodos del mismo. Al contrario que las listas enlazadas, los árboles binarios no tienen realmente un primer valor, un segundo valor, un tercer valor, etc. Se puede afirmar que el nodo

raíz viene el primero, pero, ¿quién viene a continuación? Existen diferentes métodos de recorrido de árbol ya que la mayoría de las aplicaciones con árboles son bastante sensibles al orden en el que se visitan los nodos, de forma que será preciso elegir cuidadosamente el tipo de recorrido.

El **recorrido de un árbol binario** requiere que cada nodo del árbol sea procesado (visitado) una vez, y sólo una, en una secuencia predeterminada. Existen dos enfoques generales para la secuencia de recorrido, profundidad y anchura.

En el **recorrido en profundidad,** el proceso exige un camino desde la raíz a través de un hijo, al descendiente más lejano del primer hijo antes de proseguir a un segundo hijo. En otras palabras, en el recorrido en profundidad, todos los descendientes de un hijo se procesan antes del siguiente hijo.

En el **recorrido en anchura,** el proceso se realiza horizontalmente desde el raíz a todos sus hijos; a continuación, a los hijos de sus hijos y así sucesivamente hasta que todos los nodos han sido procesados. En el recorrido en anchura, cada nivel se procesa totalmente antes de que comience el siguiente nivel.

> ***Definición***
>
> El **recorrido** de un árbol supone visitar cada nodo sólo una vez.

Dado un árbol binario que consta de raíz, un subárbol izquierdo y un subárbol derecho, se pueden definir tres tipos de secuencia de recorrido en profundidad. Estos recorridos estándar se muestran en la Figura 13.25.

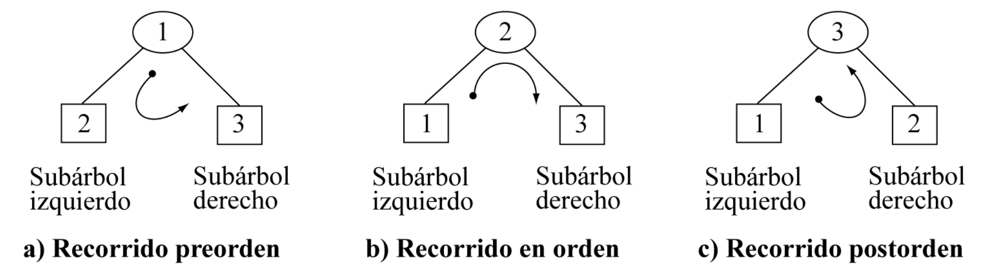


Figura 13.25 Recorridos de árboles binarios

La designación tradicional de los recorridos utiliza un nombre para el nodo raíz **(N)**, para el subárbol izquierdo **(I)** y para el subárbol derecho **(D)**.

13.5.1. Recorrido *preorden*

El recorrido *preorden*[2] **(NID)** conlleva los siguientes pasos, en los que el nodo raíz va antes que los subárboles:

1. Visitar el nodo raíz **(N)**.
2. Recorrer el subárbol izquierdo **(I)** en *preorden*.
3. Recorrer el subárbol derecho **(D)** en *preorden*.

---

[2] El nombre *preorden*, viene del prefijo latino *pre* que significa "ir antes".

Dadas las características recursivas de los árboles, el algoritmo de recorrido tiene naturaleza recursiva. Primero se procesa la raíz; a continuación, el subárbol izquierdo y, posteriormente, el subárbol derecho. Para procesar el subárbol izquierdo, se siguen los mismos pasos: raíz, subárbol izquierdo y subárbol derecho (proceso recursivo). Luego se hace lo mismo con el subárbol derecho.

Regla

En el recorrido *preorden,* el raíz se procesa antes que los subárboles izquierdo y derecho.

Si utilizamos el recorrido preorden del árbol de la Figura 13.26, se visita primero la raíz (nodo A); a continuación, se visita el subárbol izquierdo de A, que consta de los nodos B, D y E. Dado que el subárbol es a su vez un árbol, se visitan los nodos utilizando el mismo orden (NID). Por consiguiente, se visita primero el nodo B, después D (izquierdo) y por último E (derecho).

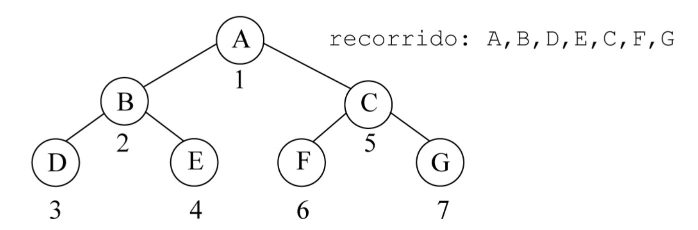


Figura 13.26 Recorrido *preorden* de un árbol binario

A continuación se visita el subárbol derecho de A, que es un árbol que contiene los nodos C, F y G. De nuevo, siguiendo el mismo orden (NID), se visita primero el nodo C, a continuación F (izquierdo) y, por último, G (derecho). En consecuencia, el orden del recorrido *preorden* para el árbol de la Figura 13.26 es A-B-D-E-C-F-G.

13.5.2. Recorrido *en orden*

El recorrido *en orden (inorder)* procesa primero el subárbol izquierdo, después el raíz y, a continuación, el subárbol derecho. El significado de *in* es que la raíz se procesa entre los subárboles.

Si el árbol no está vacío, el método implica los siguientes pasos:

1. Recorrer el subárbol izquierdo **(I)** en orden.
2. Visitar el nodo raíz **(N)**.
3. Recorrer el subárbol derecho **(D)** en orden.

En el árbol de la Figura 13.27, los nodos se han numerado en el orden en que son visitados durante el recorrido *en orden*. El primer subárbol recorrido es el subárbol izquierdo del nodo raíz (árbol cuyo nodo contiene la letra B). Este subárbol es, a su vez, otro árbol con el nodo B como raíz, por lo que siguiendo el orden IND, se visita primero D, a continuación B (nodo raíz) y, por último, E (derecha). Después, se visita el nodo raíz, A. Por último se visita el subárbol derecho de A, siguiendo el orden IND se visita primero F, después C (nodo raíz) y por último G. Por consiguiente, el orden del recorrido en orden del árbol de la Figura 13.27 es D-B-E-A-F-C-G.

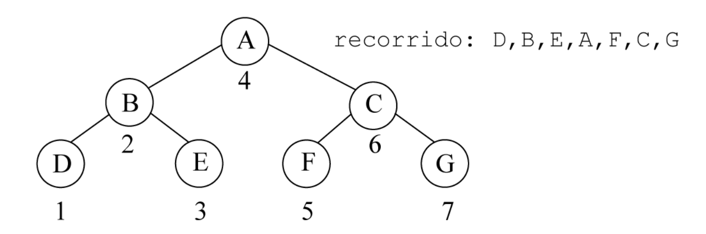


Figura 13.27 Recorrido *enorden* de un árbol binario

13.5.3. Recorrido *postorden*

El recorrido postorden **(IDN)** procesa el nodo raíz (*post*) después de que los subárboles izquierdo y derecho se hayan procesado. Comienza situándose en la hoja más a la izquierda y se procesa. A continuación, se procesa su subárbol derecho. Por último, se procesa el nodo raíz. Las etapas del algoritmo, si el árbol no está vacío, son:

1. Recorrer el subárbol izquierdo **(I)** en *postorden.*
2. Recorrer el subárbol derecho **(D)** en *postorden.*
3. Visitar el nodo raíz **(N)**.

Si se utiliza el recorrido postorden del árbol de la Figura 13.27, se visita primero el subárbol izquierdo de `A`. Este subárbol consta de los nodos `B, D` y `E`, y siguiendo el orden `IDN`, se visitará primero `D` (izquierdo), luego `E` (derecho) y, por último, `B` (nodo). A continuación, se visita el subárbol derecho de `A` que consta de los nodos `C, F y G`. Siguiendo el orden `IDN` para este árbol, se visita primero `F` (izquierdo), después `G` (derecho) y, por último, `C` (nodo). Finalmente se visita el nodo raíz, `A`. Resumiendo, el orden del recorrido *postorden* del árbol de la Figura 13.27 es `D-E-B-F-G-C-A`.

Ejercicio 13.1

Deducir el orden de los elementos en cada uno de los tres recorridos fundamentales de los árboles binarios siguientes.

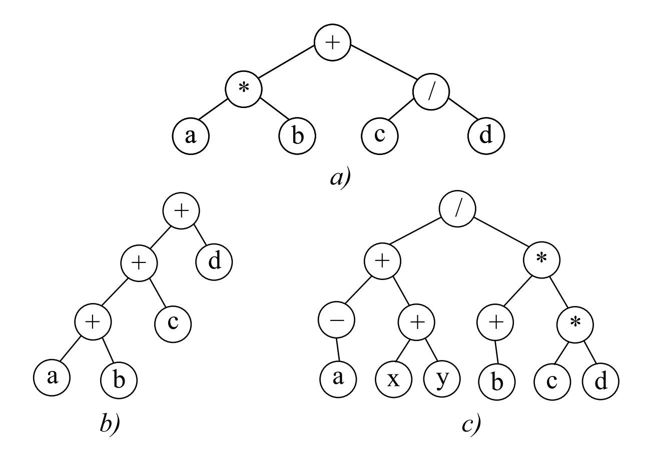


Figura 13.28 Árboles de expresión

Los elementos de los árboles binarios listados en preorden, enorden y postorden.

	Árbol a	Árbol b	Árbol c
preorden	`+*ab/cd`	`+++abcd`	`/+-a+xy*+b*cd`
enorden	`a*b+c/d`	`a+b+c+d`	`-a+x+y/+b*c*d`
postorden	`ab*cd/+`	`ab+c+d+`	`a-xy++b+cd**/`

13.5.4. Implementación

Los recorridos de un árbol binario se han definido recursivamente, los métodos que lo implementan es natural que tengan naturaleza recursiva. Prácticamente, todo consiste en trasladar la definición a codificación. Los métodos se declaran en la clase `ArbolBinario`, y tienen como argumento el nodo raíz del subárbol que se recorre; el *caso base,* para detener la recursión, es que el subárbol esté vació `(raiz == null)`.

```
// Recorrido de un árbol binario en preorden

public static void preorden(Nodo r)
{
   if (r != null)
   {
      r.visitar();
      preorden (r.subarbolIzdo());
      preorden (r.subarbolDcho());
   }
}
// Recorrido de un árbol binario en inorden
public static void inorden(Nodo r)
{
   if (r != null)
   {
      inorden (r.subarbolIzdo());
      r.visitar();
      inorden (r.subarbolDcho());
   }
}
// Recorrido de un árbol binario en postorden
public static void postorden(Nodo r)
{
   if (r != null)
   {
      postorden (r.subarbolIzdo());
      postorden (r.subarbolDcho());
      r.visitar();
   }
}
```

Nota de programación

La visita al nodo se representa mediante la llamada al método de `Nodo, visitar()`. ¿Qué hacer en el método? Depende de la aplicación que se esté realizando. Si simplemente se quieren listar los nodos, puede emplearse la siguiente sentencia:

```
void visitar()
{
   System.out.print(dato + " ");
}
```

Ejemplo 13.7

Dado un árbol binario, determinar el número de nodos de que consta.

El número de nodos se calcula recorriendo el árbol, y cada vez que se pasa por un nodo se incrementa la cuenta. El planteamiento es recursivo: el numero de nodos es `1` (nodo raíz) más el número de nodos de los subárboles izquierdo y derecho. El número de nodos de un árbol vacío es `0`, que será *el caso base* de la recursividad.

```
public static int numNodos(Nodo raiz)
{
   if (raiz == null)
      return 0;
   else
      return 1 + numNodos(raiz.subarbolIzdo()) +
                 numNodos(raiz.subarbolDcho());
}
```

El método accede a cada nodo del árbol, por lo que tiene complejidad lineal, *O(n)*.

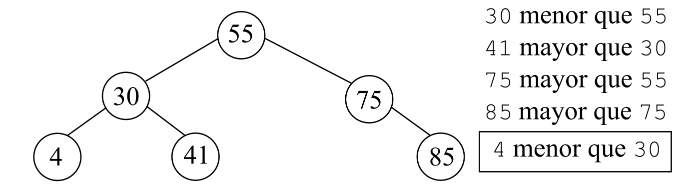
13.6. ÁRBOL BINARIO DE BÚSQUEDA

Los árboles estudiados hasta ahora no tienen un orden definido; sin embargo, los árboles binarios ordenados tienen sentido. Estos árboles se denominan árboles binarios de búsqueda, debido a que se puede buscar en ellos un término utilizando un algoritmo de búsqueda binaria similar al empleado en *arrays.*

Un **árbol binario de búsqueda** es aquel en que, dado un nodo, todos los datos del subárbol izquierdo son menores que los datos de ese nodo, mientras que todos los datos del subárbol derecho son mayores que sus propios datos. El árbol binario del Ejemplo 13.8 es de búsqueda.

Ejemplo 13.8

Árbol binario de búsqueda para nodos con el campo de datos de tipo int.

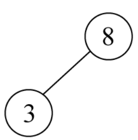


13.6.1. Creación de un árbol binario de búsqueda

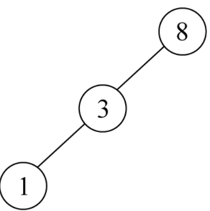
Se crea un árbol binario de búsqueda con los valores `8, 3, 1, 20, 10, 5, 4`. Para todo nodo del árbol, los datos a su izquierda deben ser menores que el dato del nodo actual, mientras que todos los datos a la derecha deben ser mayores que el del nodo actual. Inicialmente, el árbol está vacío y se desea insertar el `8`. La única elección es almacenar el `8` en el raíz:



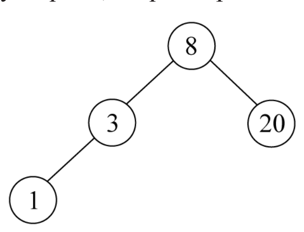
A continuación viene el `3`. Ya que `3` es menor que `8`, el `3` debe ir en el subárbol izquierdo.



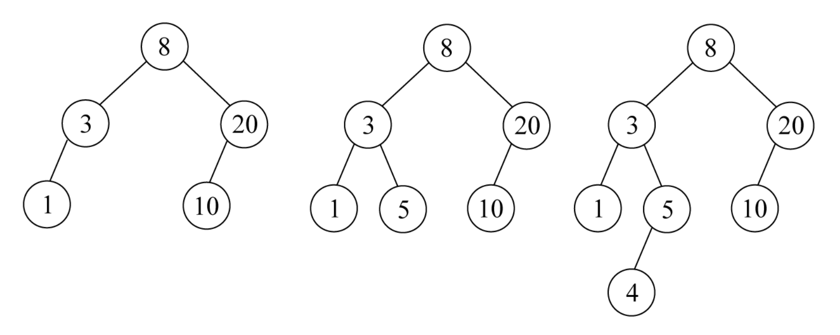
A continuación se ha de insertar `1`, que es menor que `8` y que `3`, por consiguiente, irá a la izquierda y debajo de `3`.



El siguiente número es `20`, mayor que `8`, lo que implica debe ir a la derecha de `8`.



Cada nuevo elemento se inserta como una *hoja* del árbol. Los restantes elementos se pueden situar fácilmente.

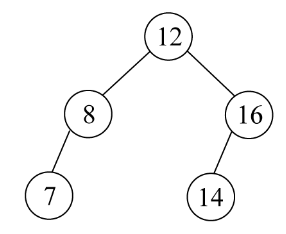


Una propiedad de los árboles binarios de búsqueda es que no son únicos para los mismos datos.

Ejemplo 13.9

Construir un árbol binario para almacenar los datos 12, 8, 7, 16 y 14.

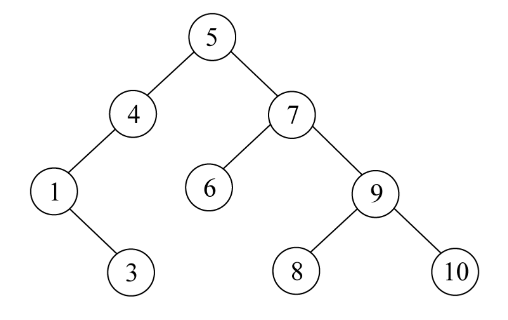
Solución



Ejemplo 13.10

Construir un árbol binario de búsqueda que corresponda a un recorrido `enorden` *cuyos elementos son: 1, 3, 4, 5, 6, 7, 8, 9 y 10*

Solución



13.6.2. Nodo de un árbol binario de búsqueda

Un nodo de un árbol binario de búsqueda no difiere en nada de los nodos de un árbol binario, tiene un campo de datos y dos enlaces a los subárboles izquierdo y derecho respectivamente. Al ser un árbol ordenado, los datos deben implementar la *interfaz* `Comparador`:

```
package arbolBinarioOrdenado;
public interface Comparador
{
   boolean igualQue(Object q);
   boolean menorQue(Object q);
   boolean menorIgualQue(Object q);
   boolean mayorQue(Object q);
   boolean mayorIgualQue(Object q);
}
```

Un árbol de búsqueda se puede utilizar cuando se necesita encontrar la información rápidamente. Un ejemplo de árbol binario de búsqueda es aquel cuyo nodo contiene información relativa a un estudiante. Cada nodo almacena el nombre del estudiante y el número de matrícula en su universidad (dato entero), que puede ser el utilizado para ordenar.

Declaración de tipos

```
class Estudiante implements Comparador
{
   int numMat;
   String nombre;

   public boolean menorQue(Object op2)
   {
        Estudiante p2 = (Estudiante) op2;
      return  numMat <  p2.numMat;
   }
   //...

}
```

13.7. OPERACIONES EN ÁRBOLES BINARIOS DE BÚSQUEDA

Los árboles binarios de búsqueda, al igual que los árboles binarios, tienen naturaleza recursiva y, en consecuencia, las operaciones sobre los árboles son recursivas, si bien siempre se tiene la opción de realizarlas de forma iterativa. Estas operaciones son:

- *Búsqueda* de un nodo. Devuelve la referencia al nodo del árbol o `null`.
- *Inserción* de un nodo. Crea un nodo con su dato asociado y lo añade, en orden, al árbol.
- *Borrado* de un nodo. Busca el nodo del árbol que contiene un dato y lo quita. El árbol debe seguir siendo de búsqueda.
- *Recorrido* de un árbol. Los mismos recorridos de un árbol binario *preorden*, *inorden* y *postorden*.

La clase `ArbolBinarioBusqueda` implementa estas operaciones. Se considera que es una *extensión* (herencia) de `ArbolBinario`.

```
package arbolBinarioOrdenado;
import arbolBinario.*;

public class ArbolBinarioBusqueda extends ArbolBinario
{
   public ArbolBinarioBusqueda()
   {
      super();
   }
   //...
```

13.7.1. Búsqueda

La búsqueda de un nodo comienza en el nodo raíz y sigue estos pasos:

1. La clave buscada se compara con la clave del nodo raíz.
2. Si las claves son iguales, la búsqueda se detiene.
3. Si la clave buscada es mayor que la clave raíz, la búsqueda se reanuda en el subárbol derecho. Si la clave buscada es menor que la clave raíz, la búsqueda se reanuda con el subárbol izquierdo.

> ***A recordar***
>
> En los árboles binarios ordenados, la búsqueda de una clave da lugar a un *camino de búsqueda,* de tal forma que *baja* por la rama izquierda si la clave buscada es menor que la clave de la raíz o *baja* por la rama derecha si la clave es mayor.

Implementación

La interfaz de la operación es el método `buscar()`con la referencia al dato (debe implementar a `Comparador`) que se busca. La búsqueda del nodo la realiza el método `localizar()`, que recibe la referencia a la raíz del subárbol y el dato; el algoritmo de búsqueda es el siguiente:

1. Si el nodo raíz contiene el dato buscado, la tarea es fácil: el resultado es, simplemente, su referencia y termina el algoritmo.
2. Si el árbol no está vacío, el subárbol específico por donde proseguir depende de que el dato requerido sea menor o mayor que el dato del raíz.
3. El algoritmo termina si el árbol está vacío, en cuyo caso devuelve `null`.

El método `localizar()` se implementa recursivamente, autollamándose con la referencia al subárbol *izquierdo* o *derecho.*

```
public Nodo buscar(Object buscado)
{
   Comparador dato;
   dato = (Comparador) buscado;
   if (raiz == null)
      return null;
   else
      return localizar(raizArbol(), dato);
}

protected Nodo localizar(Nodo raizSub, Comparador buscado)
{
   if (raizSub == null)
      return null;
   else if (buscado.igualQue(raizSub.valorNodo()))
      return raiz;
   else if (buscado.menorQue(raizSub.valorNodo()))
      return localizar(raizSub.subarbolIzdo(), buscado);
   else
      return localizar (raizSub.subarbolDcho(), buscado);
}
```

El Ejemplo 13.11 implementa la operación de búsqueda con un esquema iterativo.

Ejemplo 13.11

Aplicar el algoritmo de búsqueda de un nodo en un árbol binario ordenado para implementar la operación buscar iterativamente.

El método se codifica con un bucle cuya condición de parada es que se encuentre el nodo, o bien que el *camino de búsqueda* haya finalizado (subárbol vacío, `null`).

```
public Nodo buscarIterativo (Object buscado)
{
   Comparador dato;
   boolean encontrado = false;
   Nodo raizSub = raiz;
   dato = (Comparador) buscado;
   while (!encontrado && raizSub != null)
   {
      if (dato.igualQue(raizSub.valorNodo()))
         encontrado = true;
      else if (dato.menorQue(raizSub.valorNodo()))
         raizSub = raizSub.subarbolIzdo();
      else
         raizSub = raizSub.subarbolDcho();
   }
   return raizSub;
}
```

13.7.2. Insertar un nodo

Para añadir un nodo al árbol, se sigue el camino de búsqueda y, al final del camino, se enlaza el nuevo nodo; por consiguiente, siempre se inserta como hoja del árbol. El árbol que resulta después de insertar el nodo sigue siendo de búsqueda.

En esencia, el algoritmo de inserción se apoya en la búsqueda de un elemento, de modo que si se encuentra el elemento buscado, no es necesario hacer nada; en caso contrario, se inserta el nuevo elemento justo en el lugar donde ha acabado la búsqueda (es decir, en el lugar donde habría estado en el caso de existir).

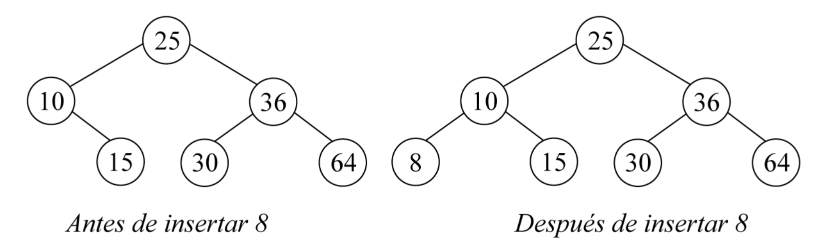


Figura 13.29 Inserción en un árbol binario de búsqueda

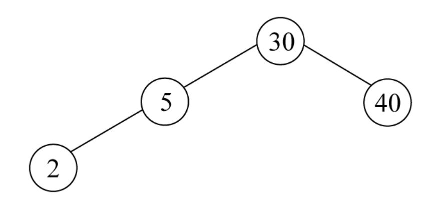
Por ejemplo, al árbol de la Figura 13.29 se le inserta el nodo `8`. El proceso describe un *camino de búsqueda* que comienza en la raíz `25`; el nodo `8` debe estar en el subárbol izquierdo de `25 (8 < 25)`. El nodo `10` es la raíz del subárbol actual, el nodo `8` debe estar en el subárbol izquierdo `(8 < 10)`, que está actualmente vacío y, por tanto, ha terminado el *camino de búsqueda*. El nodo `8` se enlaza como hijo izquierdo del nodo `10`.

A recordar

La inserción de un nuevo nodo en un árbol de búsqueda siempre se hace como nodo hoja. Para ello, se *baja* por el árbol según el *camino de búsqueda.*

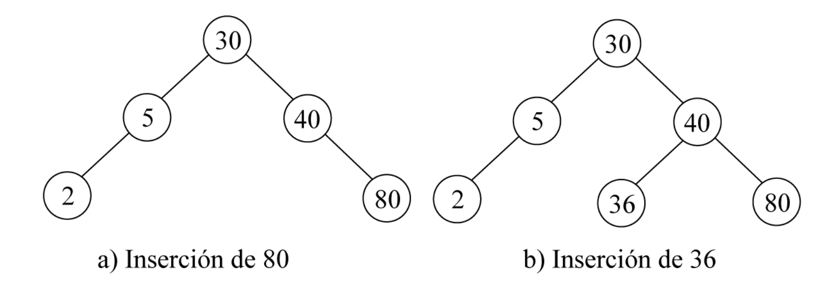
Ejemplo 13.12

Insertar un elemento con clave 80 en el árbol binario de búsqueda siguiente:



A continuación, insertar un elemento con clave 36 en el árbol binario de búsqueda resultante.

Solución



a) Inserción de 80 b) Inserción de 36

Implementación

El método `insertar()` es la interfaz de la operación, llama al método recursivo que realiza la operación y devuelve la raiz del nuevo árbol. A este método interno se le pasa la raíz actual, a partir de la cual describe el *camino de búsqueda,* y, al final, se enlaza. En un árbol binario de búsqueda no hay nodos duplicados; por ello, si se encuentra un nodo igual que el que se desea insertar, se lanza un excepción.

```
public void insertar (Object valor )throws Exception
{
   Comparador dato;
   dato = (Comparador) valor;
   raiz = insertar(raiz, dato);
}

//método interno para realizar la operación
protected Nodo
insertar(Nodo raizSub, Comparador dato) throws Exception
{
   if (raizSub == null)
      raizSub = new Nodo(dato);
   else if (dato.menorQue(raizSub.valorNodo()))
   {
      Nodo iz;
      iz = insertar(raizSub.subarbolIzdo(), dato);
      raizSub.ramaIzdo(iz);
   }
```

```
    else if (dato.mayorQue(raizSub.valorNodo()))
    {
       Nodo dr;
       dr = insertar(raizSub.subarbolDcho(), dato);
       raizSub.ramaDcho(dr);
    }
    else
       throw new Exception("Nodo duplicado");
    return raizSub;
 }
```

13.7.3. Eliminar un nodo

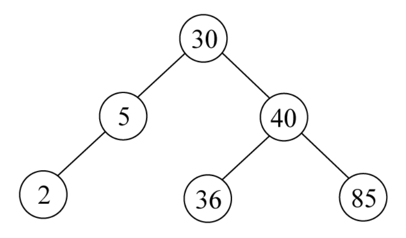
La operación de *eliminación* de un nodo es también una extensión de la operación de búsqueda, si bien más compleja que la inserción, debido a que el nodo a suprimir puede ser cualquiera y la operación debe mantener la estructura de árbol binario de búsqueda después de quitar el nodo. Los pasos a seguir son:

1. Buscar en el árbol para encontrar la posición del nodo a eliminar.
2. Si el nodo a suprimir tiene menos de dos hijos, reajustar los enlaces de su antecesor.
3. Si el nodo tiene dos hijos (rama izquierda y derecha), es necesario subir a la posición que éste ocupa el dato más próximo de sus subárboles (el inmediatamente superior o el inmediatamente inferior) con el fin de mantener la estructura de árbol binario de búsqueda.

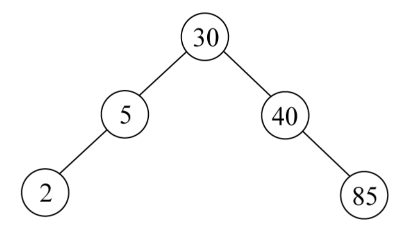
Los ejemplos 13.13 y 13.14 muestran estas dos circunstancias. El primero elimina un nodo sin descendientes, el segundo elimina un nodo que, a su vez, es la raíz de un árbol con dos ramas no vacías.

Ejemplo 13.13

Suprimir el elemento de clave 36 del siguiente árbol binario de búsqueda:

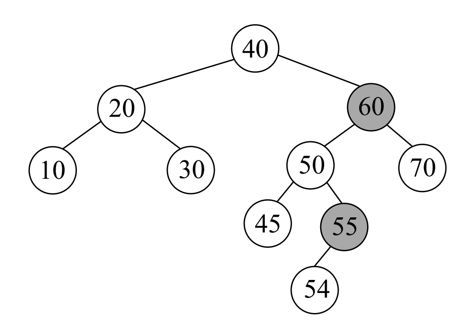


El nodo del árbol donde se encuentra la clave 36 es una hoja, por ello simplemente se reajustan los enlaces del nodo precedente en el camino de búsqueda. El árbol resultante es:

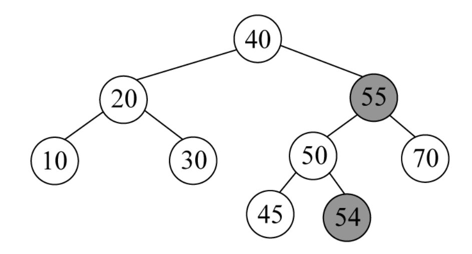


Ejemplo 13.14

Borrar el elemento de clave 60 del siguiente árbol:



Se reemplaza `60` por el elemento mayor `(55)` en su subárbol izquierdo o por el elemento más pequeño `(70)` en su subárbol derecho. Si se opta por reemplazar por el mayor del subárbol izquierdo, se mueve el `55` a la raíz del subárbol y se reajusta el árbol.



Implementación

El método `eliminar()` es la interfaz de la operación, se le pasa el elemento que se va a buscar en el árbol para retirar su nodo; llama al método sobrecargado, privado, `eliminar()` con la raíz del árbol y el elemento.

Lo primero que hace el método es buscar el nodo, siguiendo el *camino de búsqueda*. Una vez encontrado, se presentan dos casos claramente diferenciados. El primero, si el nodo a eliminar es una hoja o tiene un único descendiente, resulta una tarea fácil, ya que lo único que hay que hacer es asignar al enlace del nodo *padre* (según el *camino de búsqueda*) el descendiente del nodo a eliminar. El segundo caso, que el nodo tenga las dos ramas no vacía, exige, para mantener la estructura de árbol de búsqueda, reemplazar el dato del nodo por la *mayor de las claves menores* en el subárbol (otra posible alternativa es reemplazar el dato del nodo por la *menor de las claves mayores*). Como las claves menores están en la rama izquierda, se *baja* al primer nodo de la rama izquierda y se continúa *bajando* por las ramas derecha (claves mayores) hasta alcanzar el nodo hoja. Éste es el mayor de los menores, que reemplaza al del nodo a eliminar. El método `reemplazar()` realiza la tarea descrita.

```
public void eliminar (Object valor) throws Exception
{
   Comparador dato;
   dato = (Comparador) valor;
   raiz = eliminar(raiz, dato);
```

```
}

//método interno para realizar la operación
protected Nodo
eliminar (Nodo raizSub, Comparador dato) throws Exception
{
  if (raizSub == null)
    throw new Exception ("No encontrado el nodo con la clave");
  else if (dato.menorQue(raizSub.valorNodo()))
{
     Nodo iz;
     iz = eliminar(raizSub.subarbolIzdo(), dato);
     raizSub.ramaIzdo(iz);
   }
   else if (dato.mayorQue(raizSub.valorNodo()))
   {
     Nodo dr;
     dr = eliminar(raizSub.subarbolDcho(), dato);
     raizSub.ramaDcho(dr);
   }
   else                           // Nodo encontrado
   {
     Nodo q;
     q = raizSub;                 // nodo a quitar del árbol
     if (q.subarbolIzdo() == null)
       raizSub = q.subarbolDcho();
     else if (q.subarbolDcho() == null)
       raizSub = q.subarbolIzdo();
     else
     {                            // tiene rama izquierda y derecha
       q = reemplazar(q);
     }
     q = null;
   }
   return raizSub;
}

// método interno para susutituir por el mayor de los menores
private Nodo reemplazar(Nodo act)
{
   Nodo a, p;

   p = act;
   a = act.subarbolIzdo();          // rama de nodos menores
   while (a.subarbolDcho() != null)
   {
     p = a;
     a = a.subarbolDcho();
   }
   act.nuevoValor(a.valorNodo());
   if (p == act)
     p.ramaIzdo(a.subarbolIzdo());
   else
     p.ramaDcho(a.subarbolIzdo());
   return a;
}
```

RESUMEN

En este capítulo se introdujo y se desarrolló la estructura de datos dinámica árbol. Esta estructura, muy potente, se puede utilizar en una gran variedad de aplicaciones de programación.

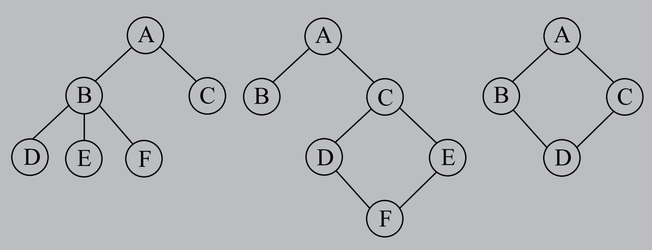
La estructura árbol más utilizada normalmente es el *árbol binario*. Un **árbol binario** es un árbol en el que cada nodo tiene como máximo dos hijos, llamados subárbol izquierdo y subárbol derecho. En un árbol binario, cada elemento tiene cero, uno o dos hijos. El nodo raíz no tiene un padre, pero cada elemento restante sí tiene un padre.

La altura de un árbol binario es el número de ramas entre el raíz y la hoja más lejana, más `1`. Si el árbol A es vacío, la altura es `0`. *El nivel* o *profundidad* de un elemento es un concepto similar al de altura.

Un árbol binario no vacío está *equilibrado totalmente* si sus subárboles izquierdo y derecho tienen la misma altura y ambos son o bien vacíos o totalmente equilibrados. Los árboles binarios presentan dos tipos característicos: *árboles binarios de búsqueda* y *árboles binarios de expresiones*. Los árboles binarios de búsqueda se utilizan, fundamentalmente, para mantener una colección ordenada de datos, y los árboles binarios de expresiones, para almacenar expresiones.

EJERCICIOS

13.1. Explicar por qué cada una de las siguientes estructuras no es un árbol binario:



13.2. Considérese el árbol siguiente,

a) ¿Cuál es su altura?

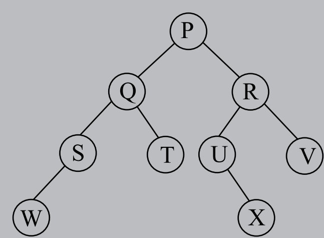
b) ¿Está el árbol equilibrado? ¿Por qué?

c) Listar todos los nodos hoja.

d) ¿Cuál es el predecesor inmediato (padre) del nodo U?

e) Listar los hijos del nodo R.

f) Listar los sucesores del nodo R.



13.3. Para cada una de las siguientes listas de letras,

a) dibujar el árbol binario de búsqueda que se construye cuando las letras se insertan en el orden dado,

b) realizar recorridos enorden, preorden y postorden del árbol y mostrar la secuencia de letras que resultan en cada caso.

(i) `M, Y, T, E, R` **(iii)** `R, E, M, Y, T`

(ii) `T, Y, M, E, R` **(iv)** `C, O, R, N, F, L, A, K, E, S`

13.4. En el árbol del Ejercicio 13.2, recorrer cada árbol utilizando los órdenes siguientes: `NDI, DNI, DIN.`

13.5. Dibujar los árboles binarios que representan las siguientes expresiones:

a) `(A+B)/(C-D)`

b) `A+B+C/D`

c) `A-(B-(C-D)/(E+F))`

d) `(A+B)*((C+D)/(E+F))`

e) `(A-B)/((C*D)-(E/F))`

13.6. El recorrido preorden de un cierto árbol binario produce

`ADFGHKLPQRWZ`

y el recorrido *enorden* produce

`GFHKDLAWRQPZ`

Dibujar el árbol binario.

13.7. Escribir un método recursivo que cuente las hojas de un árbol binario.

13.8. Escribir un método que determine el número de nodos que se encuentran en el nivel n de un árbol binario.

13.9. Escribir un método que tome un árbol como entrada y devuelva el número de hijos del árbol.

13.10. Escribir un método *booleano* al que se le pase una referencia a un árbol binario y devuelva verdadero *(true)* si el árbol es completo y falso *(false)* en caso contrario.

13.11. Se dispone de un árbol binario de elementos de tipo entero. Escribir métodos que calculen:

a) La suma de sus elementos.

b) La suma de sus elementos que son múltiplos de 3.

13.12. Diseñar un método iterativo que encuentre el número de nodos hoja en un árbol binario.

13.13. En un árbol de búsqueda cuyo campo clave es de tipo entero, escribir un método que devuelva el número de nodos cuya clave se encuentra en el rango [x1, x2].

13.14. Diseñar un método que visite los nodos del árbol por niveles; primero el nivel 0, después los nodos del nivel 1, y del nivel 2 y así hasta el último nivel.

PROBLEMAS

13.1. Se dispone de un archivo de texto en el que cada línea contiene la siguiente información

Columnas	1-20	Nombre
	21-31	Número de la Seguridad Social
	32-78	Dirección

Escribir un programa que lea cada registro de datos y lo inserte en un árbol, de modo que cuando el árbol se recorra en *inorden,* los números de la seguridad social se ordenen de forma ascendente.

13.2. Escribir un programa que lea un texto de longitud indeterminada y que produzca como resultado la lista de todas las palabras diferentes contenidas en el texto, así como su frecuencia de aparición. Hacer uso de la estructura árbol binario de búsqueda para localizar cada nodo del árbol que tenga una palabra y su frecuencia.

13.3. Escribir un programa que procese un árbol binario cuyos nodos contengan caracteres, y a partir del siguiente menú de opciones:

`I` (seguido de un carácter):	Insertar un carácter.
`B` (seguido de un carácter):	Buscar un carácter.
`RE`:	Recorrido en orden.
`RP`:	Recorrido en preorden.
`RT`:	Recorrido postorden.
`SA`:	Salir.

13.4. Escribir un método booleano `identicos()` que permita decir si dos árboles binarios son iguales.

13.5. Construir un método en la clase `ArbolBinarioBusqueda` que encuentre el nodo máximo.

13.6. Construir un método recursivo para escribir todos los nodos de un árbol binario de búsqueda cuyo campo clave sea mayor que un valor dado (el campo clave es de tipo entero).

13.7 Escribir un método que determine la altura de un nodo. Escribir un programa que cree un árbol binario con números generados aleatoriamente y muestre por pantalla:

- La altura de cada nodo del árbol.
- La diferencia de altura entre las ramas izquierda y derecha de cada nodo.

13.8. Diseñar métodos no recursivos que listen los nodos de un árbol en inorden, preorden y postorden.

13.9. Dados dos árboles binarios `A` y `B`, se dice que son *parecidos* si el árbol `A` puede ser transformado en el árbol `B` intercambiando los hijos izquierdo y derecho (de alguno de sus nodos). Escribir un método que determine si dos árboles son *parecidos.*

13.10. Dado un árbol binario de búsqueda, construir su árbol espejo. Un árbol espejo es el que se construye a partir de uno dado convirtiendo el subárbol izquierdo en subárbol derecho, y viceversa.

13.11. Un árbol binario de búsqueda puede implementarse con un *array*. La representación no enlazada correspondiente consiste en que para cualquier nodo del árbol almacenado en la posición **i** del *array,* su hijo izquierdo se encuentra en la posición **2*i** y su hijo derecho en la posición **2*i + 1**. A partir de esta representación, diseñar los métodos con las operaciones correspondientes para gestionar interactivamente un árbol de números enteros.

13.12. Dado un árbol binario de búsqueda diseñar un método que liste los nodos del árbol ordenados descendentemente.

CAPITULO 14

Árboles de búsqueda equilibrados

Objetivos

Con el estudio de este capítulo, usted podrá:

- Conocer la eficiencia de un árbol de búsqueda.
- Construir un árbol binario equilibrado conociendo el número de claves.
- Construir un árbol binario de búsqueda equilibrado.
- Describir los diversos tipos de movimientos que se hacen cuando se desequilibra un árbol.
- Diseñar y declarar la clase *ArbolEquilibrado*.

Contenido

Conceptos clave

- Altura de un árbol.
- Árbol de búsqueda.
- Camino de búsqueda.
- Complejidad logarítmica.
- Equilibrio.
- Factor de equilibrio.
- Hoja.
- Rotaciones.

INTRODUCCIÓN

En el Capítulo 13 se introdujo el concepto de árbol binario. Se utiliza un árbol binario de búsqueda para almacenar datos organizados jerárquicamente. Sin embargo, en muchas ocasiones, las inserciones y eliminaciones de elementos en el árbol no ocurren en un orden predecible: es decir, los datos no están organizados jerárquicamente.

En este capítulo se estudian tipos de árboles adicionales: los *árboles equilibrados* o *árboles AVL,* como también se les conoce, que ayudan eficientemente a resolver las situaciones citadas.

El concepto de árbol equilibrado así como los algoritmos de manipulación constituyen el motivo central de este capítulo. Los métodos que describen este tipo de árboles fueron descritos en 1962 por los matemáticos rusos G. M. Adelson-Velskii y E. M. Landis.

14.1. EFICIENCIA DE LA BÚSQUEDA EN UN ÁRBOL ORDENADO

La eficiencia de una búsqueda en un árbol binario ordenado varía entre `O(n)` y `O(log(n))`, dependiendo de la estructura que presente el árbol.

Si los elementos se añaden en el árbol mediante el algoritmo de inserción expuesto en el capítulo anterior, la estructura resultante del árbol dependerá del orden en que sean añadidos. Así, si todos los elementos se insertan en orden creciente o decreciente, el árbol tendrá todas la ramas izquierda o derecha, respectivamente, vacías. En este caso, la búsqueda en dicho árbol será totalmente secuencial.

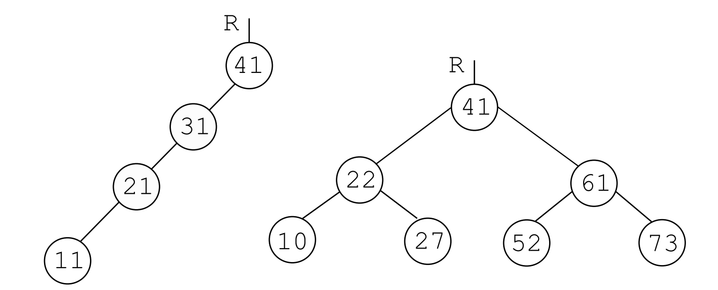


Figura 14.1 Árbol degenerado y equilibrado de búsqueda

Sin embargo, si la mitad de los elementos insertados después de otro con clave `K` tienen claves menores de `K` y la otra mitad claves mayores de `K`, se obtiene un árbol equilibrado (también llamado *balanceado)*, en el cual las comparaciones para obtener un elemento son como máximo $\texttt{Log}_2\texttt{(n)}$, para un árbol de `n` nodos.

En los árboles de búsqueda, el número promedio de comparaciones que deben realizarse para las operaciones de inserción, eliminación y búsqueda varía entre $\texttt{Log}_2\texttt{(n)}$, para el mejor de los casos y `n` para el *peor de los casos*. Para optimizar los tiempos de búsqueda en los árboles ordenados surgen los árboles *casi equilibrados,* en los que la complejidad de la búsqueda es logarítmica, `O(Log n))`.

14.2. ÁRBOL BINARIO EQUILIBRADO, ÁRBOLES AVL

Un árbol totalmente equilibrado se caracteriza porque la altura de la rama izquierda es igual que la altura de la rama derecha para cada uno de los nodos del árbol. Es un árbol ideal, pero no siempre se puede conseguir que el árbol esté totalmente balanceado.

La estructura de datos de árbol equilibrado que se utiliza es la del árbol **AVL**. El nombre es en honor de Adelson-Velskii-Landis, que fueron los primeros científicos en estudiar las propiedades de esta estructura de datos. Son árboles ordenados o de búsqueda que, además, cumplen la condición de balanceo para cada uno de los nodos.

Definición

Un árbol equilibrado o *árbol AVL* es un árbol binario de búsqueda en el que las alturas de los subárboles izquierdo y derecho de cualquier nodo difieren como máximo en 1.

La Figura 14.2 muestra dos árboles de búsqueda. El de la izquierda está equilibrado: el de la derecha es el resultado de insertar la clave `2` en el anterior, según el algoritmo de inserción en árboles de búsqueda. La inserción provoca que se *viole* la condición de equilibrio en el nodo raíz del árbol.

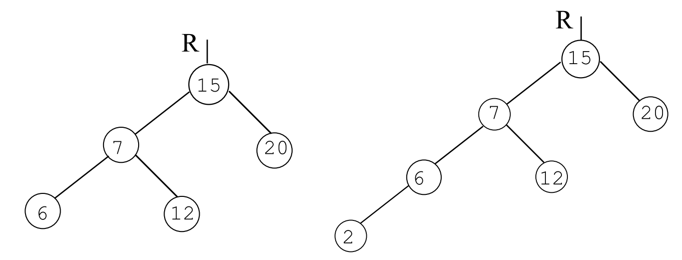


Figura 14.2 Dos árboles de búsqueda; el de la izquierda equilibrado, el otro no

La condición de equilibrio de cada nodo implica una restricción en las alturas de los subárboles de un árbol AVL. Si v_r es la raíz de cualquier subárbol de un árbol equilibrado y `h` la altura de la rama izquierda, entonces la altura de la rama derecha puede tomar los valores: `h-1, h, h+1` y ello aconseja asociar a cada nodo el parámetro denominado *factor de equilibrio* o *balance de un nodo*. Se define como la altura del subárbol derecho menos la altura del subárbol izquierdo correspondiente. El factor de equilibrio de cada nodo en un árbol equilibrado puede tomar los valores: `1, -1` ó `0`. La Figura 14.3 muestra un árbol balanceado con el factor de equilibrio de cada nodo.

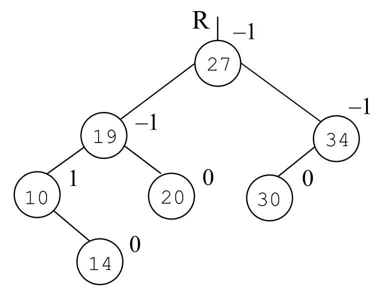


Figura 14.3 Árbol equilibrado con el factor de equilibrio de cada nodo

A tener en cuenta

La altura o profundidad de un árbol binario es el nivel máximo de sus hojas más uno. La altura de un árbol nulo se considera cero.

14.2.1. Altura de un árbol equilibrado, árbol AVL

No resulta fácil determinar la altura promedio de un árbol AVL, por ello se determina la altura en el *peor de los casos*, es decir, la altura máxima que puede tener un árbol equilibrado con un número de nodos `n`. La altura es un parámetro importante ya que coincide con el número de iteraciones que se realizan para *bajar* desde el nodo raíz al nivel mas profundo de las hojas. La eficiencia de los algoritmos de búsqueda, inserción y borrado depende de la altura del árbol AVL.

Para calcular la altura máxima de un árbol AVL de `n` nodos se parte del siguiente razonamiento: *¿Cuál es el número mínimo de nodos que puede tener un árbol binario para que se considere equilibrado con una altura h?* Si ese árbol es A_h, tendrá dos subárboles izquierdo y derecho respectivamente, A_i y A_d cuya altura difiere en `1`, supongamos que tienen de altura `h-1` y `h-2` respectivamente. Al considera que A_h es el árbol de menor número de nodos de altura `h`, entonces A_i y A_d también son árboles AVL de menor número de nodos, pero de altura `h-1` y `h-2` y son designados como A_{h-1} y A_{h-2}. Este razonamiento se sigue extendiendo a cada subárbol y se obtienen árboles equilibrados como los de la Figura 14.4.

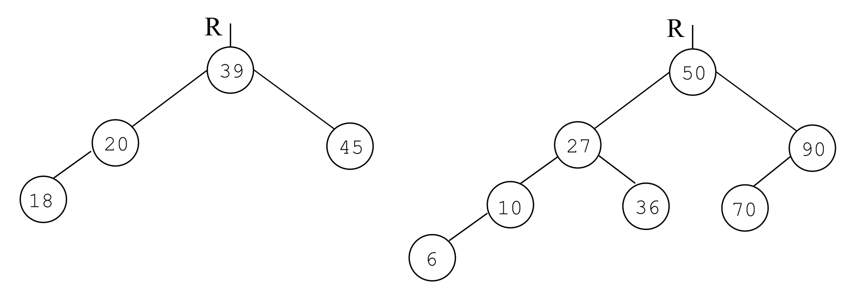


Figura 14.4 Árboles de Fibonacci

La construcción de árboles binarios equilibrados siguiendo esta estrategia tiene la siguiente representación matemática:

$$A_h = A_{h-1} + A_{h-2}$$

La expresión matemática expuesta es similar a la ley de recurrencia que permite encontrar números de Fibonacci, $a_n = a_{n-1} + a_{n-2}$. Por esa razón, a los árboles equilibrados construidos con esta ley de formación se les conoce como *árboles de Fibonacci*.

El objetivo que se persigue es encontrar el número de nodos, `n`, que hace la altura máxima. El número de nodos de un árbol es la suma de los nodos de su rama izquierda, rama derecha más uno (la raíz). Si ese número es N_h se puede escribir:

$$N_h = N_{h-1} + N_{h-2} + 1$$

donde $N_0 = 1$, $N_1 = 2$, $N_2 = 4$, $N_3 = 7$, y así sucesivamente. Se observa que los números $N_h + 1$ cumplen la definición de los números de Fibonacci. El estudio matemático de la función generadora de los números de Fibonacci permite encontrar esta relación:

$$N_h + 1 \approx \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{h+3}$$

Tomando logaritmos se encuentra la altura `h` en función del número de nodos, N_h:

$$h \approx 1.44 \log (N_h)$$

Como conclusión, el árbol equilibrado de `n` nodos menos denso tiene como altura `1.44log n`, donde `n` es el número de nodos en el *peor de los casos* del árbol AVL de altura `h` y se puede afirmar que la complejidad de una búsqueda es `O(Log n)`.

A tener en cuenta

La altura de un árbol binario perfectamente equilibrado de `n` nodos es `log n`. Las operaciones que se aplican a los árboles AVL no requieren más del 44% de tiempo (en el caso más desfavorable) que si se aplican a un árbol perfectamente equilibrado.

Ejercicio 14.1

Se tienen n claves que se van a organizar jerárquicamente formando un árbol equilibrado. Escribir un programa para formar el árbol AVL siguiendo la estrategia de los árboles de Fibonacci descrita en la sección anterior.

La formación de un árbol balanceado de `n` nodos se parece mucho a la secuencia de los números de Fibonacci:

```
a(n) = a(n-2) + a(n-1)
```

Un *árbol de Fibonacci* (árbol equilibrado) se define como:

1. Un árbol vacío es el árbol de Fibonacci de altura `0`.
2. Un nodo único es un árbol de Fibonacci de altura `1`.
3. Si A_{h-1} y A_{h-2} son árboles de Fibonacci de alturas `h-1` y `h-2`, entonces

   $A_h = < A_{h-1} , x, A_{h-2} >$ es árbol de Fibonacci de altura `h`.

El número de nodos, A_h , viene dado por la sencilla relación recurrente:

$$N_0 = 0$$
$$N_1 = 1$$
$$N_h = N_{h-1} + 1 + N_{h-2}$$

Para conseguir un árbol AVL con un número dado, `n`, de nodos hay que distribuir equitativamente los nodos a la izquierda y a la derecha de un nodo dado. En definitiva, se trata de seguir la relación de recurrencia anterior, que se expresa recursivamente, consiste en:

1. Crear nodo raíz.
2. Generar el subárbol izquierdo con $n_i = n/2$ nodos del nodo *raíz* utilizando la misma estrategia.
3. Generar el subárbol derecho con $n_d = n - n_i - 1$ nodos del nodo raíz utilizando la misma estrategia.

En este ejercicio el árbol no va a ser de búsqueda, simplemente un árbol binario de números enteros, generados aleatoriamente.

```
import java.io.*;
import java.util.*;
import arbolBinario.*;  // clase Nodo y ArbolBinario
```

```
public class ArbolDeFibonacci
{
   static final int TOPEMAX = 29;

   public static void main(String [] a)throws Exception
   {
      ArbolBinario arbolFib;
      int n;
      BufferedReader entrada = new BufferedReader(
                         new InputStreamReader(System.in));
      do {
         System.out.print("Número de nodos del árbol: ");
         n = Integer.parseInt(entrada.readLine());
         } while (n <= 0);

      arbolFib = new ArbolBinario(arbolFibonacci(n));
      System.out.println(«Árbol de Fibonacci de máxima altura:»);
      dibujarArbol(arbolFib.raizArbol(), 0);
   }
   // método recursivo que genera el árbol de Fibonacci
   public static Nodo arbolFibonacci(int n)
   {
      int nodosIz, nodosDr;
      Integer clave;
      Nodo nuevoRaiz;

      if (n == 0)
         return null;
      else
      {
         nodosIz = n / 2;
         nodosDr = n - nodosIz - 1;
            // nodo raíz con árbol izquierdo y derecho de Fibonacci
         clave = new Integer((int)(Math.random()*TOPEMAX) + 1);
         nuevoRaiz = new Nodo(arbolFibonacci(nodosIz),
                              clave,
                              arbolFibonacci(nodosDr));

         return nuevoRaiz;
      }
   }
   // método de escritura de las claves del árbol
   static void dibujarArbol(Nodo r, int h)
   {
      /*
         escribe las claves del arbol de fibonacci; h estable
         una separacion entre nodos
      */
      int i;
      if (r != null)
      {
         dibujarArbol(r.subarbolIzdo(), h + 1);
         for (i = 1; i <= h; i++)
            System.out.print(" ");
         System.out.println(r.valorNodo());
         dibujarArbol(r.subarbolDcho(), h + 1);
      }
   }
}
```

14.3. INSERCIÓN EN ÁRBOLES DE BÚSQUEDA EQUILIBRADOS: ROTACIONES

Los árboles equilibrados, árboles AVL, son árboles de búsqueda y, por consiguiente, para añadir un elemento se ha de seguir el mismo algoritmo que en dichos árboles de búsqueda. Se compara la nueva clave con la clave del raíz, continúa por la rama izquierda o derecha según sea menor o mayor (describe el *camino de búsqueda*) y termina insertándose como nodo hoja. Esta operación, como ha quedado demostrado al determinar la altura en el peor de los casos, tiene una complejidad logarítmica. Sin embargo, la nueva inserción puede hacer que aumente la altura de una rama, de manera que cambie el factor de equilibrio del nodo raíz de dicha rama. Este hecho hace necesario que el algoritmo de inserción *regrese* por el *camino de búsqueda* actualizando el factor de equilibrio de los nodos. La Figura 14.5 muestra un árbol equilibrado y el mismo árbol justo después de la inserción de una nueva clave que provoca que *rompa* la condición de balanceo.

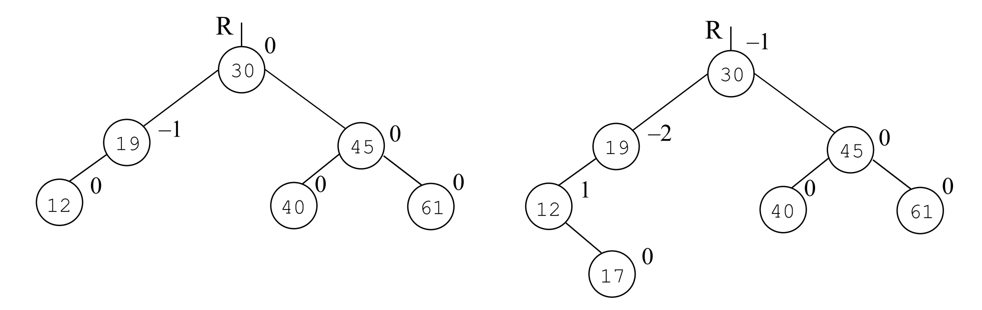


Figura 14.5 Árbol equilibrado; el mismo después de insertar la clave `17`

> ***Nota***
>
> Una inserción de una nueva clave, o un borrado, puede destruir el criterio de equilibrio de varios nodos del árbol. Se debe recuperar la condición de equilibrio del árbol antes de dar por finalizada la operación para que el árbol siga siendo equilibrado.

La estructura del nodo en un árbol equilibrado, es una *extensión* de la declarada para un árbol binario. Para determinar si el árbol está equilibrado debe manejarse información relativa al balanceo o factor de equilibrio de cada nodo. Por esa razón, se añade al nodo un campo más: el factor de equilibrio `(fe)`. Este campo puede tomar los valores: `-1, 0, +1`.

```
package arbolAVL;
import arbolBinario.*;
public class NodoAvl extends Nodo
{
   int fe;
   public NodoAvl(Object valor)
   {
      super(valor);
      fe = 0;
   }
   public NodoAvl(Object valor, NodoAvl ramaIzdo, NodoAvl ramaDcho)
   {
```

```
      super (ramaIzdo, valor, ramaDcho);
      fe = 0;
   }
}
```

Las operaciones básicas de un árbol de búsqueda equilibrado son insertar y eliminar un elemento; además, se necesitan operaciones auxiliares para mantener los criterios de equilibrio. La clase `ArbolAvl` implementa estas operaciones, su constructor inicializa la raíz a `null`, es decir árbol vacío:

```
package arbolAVL;
import arbolBinarioOrdenado.Comparador;
public class ArbolAvl
{
   NodoAvl raiz;
   public ArbolAvl()
   {
      raiz = null;
   }
   public NodoAvl raizArbol ()
   {
      return raiz;
   }
   //...
```

14.3.1. Proceso de inserción de un nuevo nodo

Inicialmente, se aplica el algoritmo de inserción en un árbol de búsqueda; este algoritmo sigue el *camino de búsqueda* hasta llegar al fondo del árbol y se enlaza como nodo *hoja* y con *factor de equilibrio* `0`. Pero el proceso no puede terminar, es necesario recorrer el *camino de búsqueda* en sentido contrario, hacia la raíz, para actualizar el campo adicional *factor de equilibrio*. Después de una inserción, sólo los nodos que se encuentran en el camino de búsqueda pueden haber cambiado el factor de equilibrio.

La actualización del *factor de equilibrio* `(fe)` puede hacer que éste *mejore*. Esto ocurre cuando un nodo está descompensado a la izquierda y se inserta el nuevo nodo en la rama izquierda; al crecer en altura dicha rama, el `fe` se hace `0`, se ha *mejorado* el equilibrio. La Figura 14.6 muestra el árbol a), en el que el nodo `90` tiene `fe = 1`; en el árbol b), después de insertar el nodo con clave `60`, el nodo `90` tiene `fe = 0`.

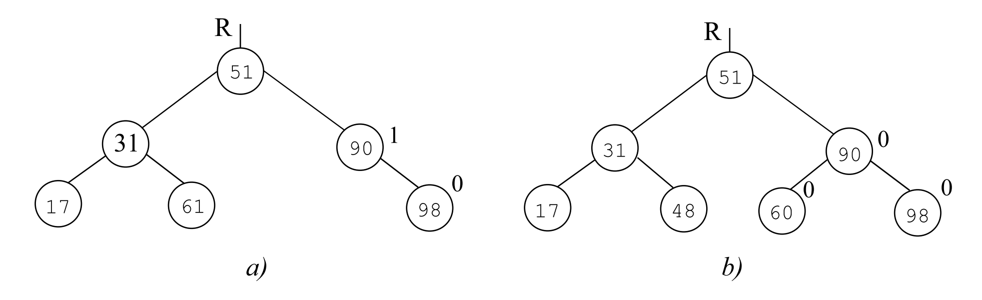


Figura 14.6 Mejora en la condición de equilibrio al insertar un nuevo nodo con clave `60`

La actualización del `fe` de un nodo del *camino de búsqueda* que, originalmente, tiene las ramas izquierda y derecha de la misma altura ($h_{Ri} = h_{Rd}$), no produce la rotura del criterio de equilibrio.

Al actualizar el nodo cuyas ramas izquierda y derecha del árbol tienen altura diferente, $|h_{Ri}-h_{Rd}|=1$, si se inserta el nodo en la rama más alta rompe el criterio de equilibrio del árbol, la diferencia de altura pasa a ser `2` y es necesario reestructurarlo.

Hay cuatro casos que se deben tener en cuenta al reestructurar un nodo `A`, según dónde se haya hecho la inserción:

1. Inserción en el subárbol izquierdo de la rama izquierda de `A`.
2. Inserción en el subárbol derecho de la rama izquierda de `A`.
3. Inserción en el subárbol derecho de la rama derecha de `A`.
4. Inserción en el subárbol izquierdo de la rama derecha de `A`.

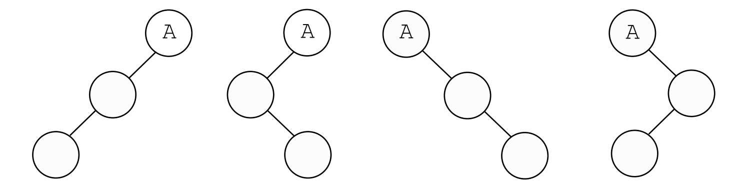


Figura 14.7 Cuatro tipos de reestructuraciones del equilibrio de un nodo

El primer y el tercer caso *(izquierda-izquierda, derecha-derecha)* se resuelven con una *rotación simple*. El segundo y el cuarto caso *(izquierda-derecha, derecha-izquierda)* se resuelven con una *rotación doble*.

La rotación simple implica a dos nodos: el nodo `A` (nodo con `|fe| = 2`) y el descendiente izquierdo o derecho según el caso. En la rotación doble están implicados tres nodos: el nodo `A`, nodo descendiente izquierdo y el descendiente derecho de éste; o bien el caso simétrico, nodo `A`, descendiente derecho y descendiente izquierdo de éste.

> ***A recordar***
>
> Una reestructuración de los nodos implicados en la violación de *criterio de equilibrio,* ya sea una rotación simple o doble, hace que se recupere el equilibrio en todo el árbol, no siendo necesario seguir analizando los nodos del *camino de búsqueda*.

El proceso de *regresar* por el *camino de búsqueda* termina cuando se llega a la raíz del árbol, o cuando se realiza la reestructuración en un nodo del mismo. Una vez realizada una reestructuración, no es necesario determinar el *factor de equilibrio* de los restantes nodos, debido a que dicho factor queda como el que tenía antes de la inserción, ya que la reestructuración hace que no aumente la altura.

14.3.2. Rotación simple

La rotación simple resuelve la *violación del equilibrio* de un nodo *izquierda-izquierda,* simétrica a la *derecha-derecha* . El árbol de la Figura 14.8*a* tiene el nodo `C` con factor de equilibrio `–1`; el árbol de la Figura 14.8*b* es el resultado de insertar el nodo `A`. Resulta que ha crecido la altura de la rama izquierda, es un desequilibrio *izquierda-izquierda* que se resuelve con una rotación simple, rotación `II`. La Figura 14.9 es el árbol resultante de la rotación, el nodo `B` se ha convertido

en la raíz, el nodo `C` en su rama derecha y el nodo `A` continúa como rama izquierda. Con estos movimientos el árbol sigue siendo de búsqueda y se equilibra.

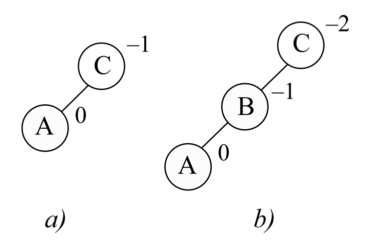


Figura 14.8 Árbol binario AVL y árbol después de insertar nueva clave por la izquierda

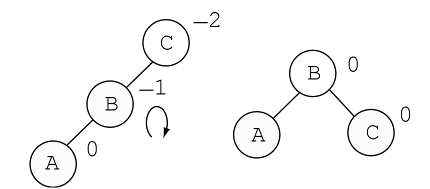


Figura 14.9 Árbol binario después de rotación simple `II`

La Figura 14.10 muestra el otro caso de violación de la condición de equilibrio que se resuelve con una rotación simple. Inicialmente, el nodo `A` tiene como factor de equilibrio `+1`, al insertar el nodo `C` el factor de equilibrio de `A` pasa a ser `+2`, ya que se ha insertado por la derecha y, por consiguiente, ha crecido la altura de la rama derecha. La Figura 14.11 muestra la resolución de este desequilibrio, una rotación simple que se puede denominar rotación `DD`. En el árbol resultante de la rotación, el nodo `B` se ha convertido en la raíz, el nodo `A` es su rama izquierda y el nodo `C` continúa como rama derecha. Con estos movimientos, el árbol sigue siendo de búsqueda y queda equilibrado.

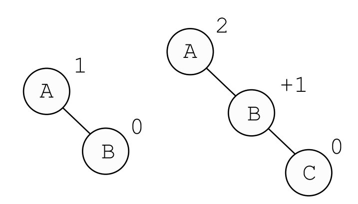


Figura 14.10 Árbol binario AVL y árbol después de insertar nueva clave por la derecha

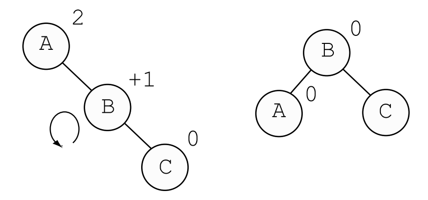


Figura 14.11 Árbol binario después de rotación simple `DD`

14.3.3. Movimiento de enlaces en la rotación simple

Los cambios descritos en la rotación simple afectan a dos nodos, el tercero no se modifica, es necesario sólo una rotación. Para la rotación simple a la *izquierda,* rotación `II`, los ajustes necesarios de los enlaces, suponiendo `n` la referencia al nodo problema y `n1` la referencia al nodo de su rama izquierda, son:

```
n.izdo  = n1.dcho
n1.dcho = n
n       = n1
```

Una vez realizada la rotación, los factores de equilibrio de los nodos que intervienen siempre son `0`, los subárboles izquierdo y derecho tienen la misma altura. Incluso la altura del subárbol implicado es la misma después de la inserción que antes. La Figura 14.12 muestra estos movimientos de los enlaces.

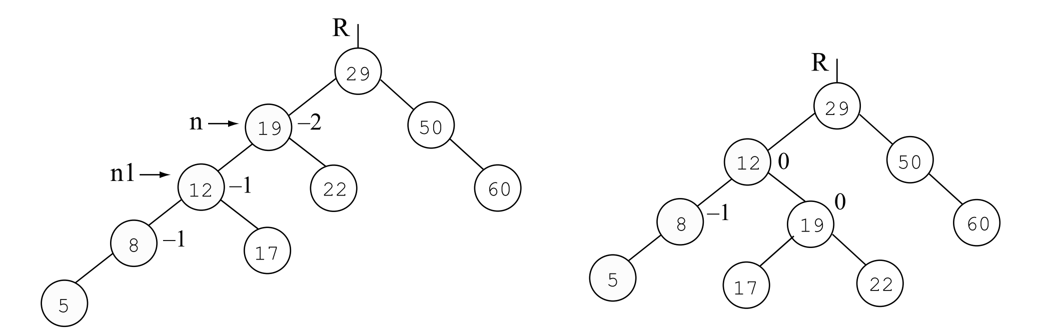


Figura 14.12 Rotación simple a izquierda en un árbol después de insertar la clave `5`

Si la rotación simple es a *derechas,* rotación `DD`, los cambios en los enlaces del nodo `n` (con factor de equilibrio `+2`) y del nodo de su rama derecha, `n1`, son:

```
n.dcho    =  n1.izdo
n1.izdo   =  n
n         =  n1
```

Realizada la rotación, los factores de equilibrio de los nodos que intervienen es `0`. Se puede observar que estos ajustes son simétricos a los realizados en la rotación `II`.

14.3.4. Rotación doble

Con la rotación simple no es posible resolver todos los casos de violación del criterio de equilibrio. El árbol de búsqueda de la Figura 14.13a está desequilibrado, con factores de equilibrio `+2, –1 y 0`. La Figura 14.13b aplica la rotación simple, rotación `DD`. La única solución consiste en *subir* el nodo `40` como raíz del subárbol, como rama izquierda situar el nodo `30` y como rama derecha el nodo `60`; la altura del subárbol resultante es la misma que antes de insertar. Se ha realizado una rotación doble, *derecha-izquierda,* en la que intervienen los tres nodos.

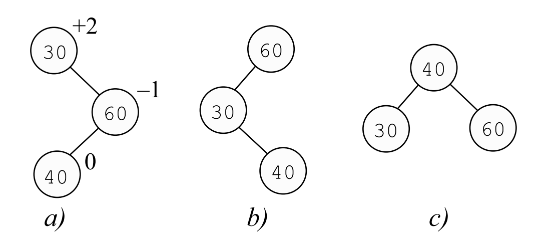


Figura 14.13 *a)* Árbol después de insertar clave `40`; *b)* rotación `DD`; *c)* rotación doble para equilibrar

La Figura 14.14*a* muestra un árbol binario de búsqueda después de insertar la clave `60`. Al *volver* por el *camino de búsqueda* para actualizar los factores de equilibrio, el nodo `75` pasa a tener `fe = -1` (se ha insertado por su izquierda), el nodo `50` pasa a tener `fe = +1` y el nodo `80` tendrá como `fe =-2`. Es un caso simétrico al descrito en la Figura 14.13, se reestablece el equilibrio con una rotación doble, simétrica con respecto a la anterior como se muestra en la Figura 14.14*b*.

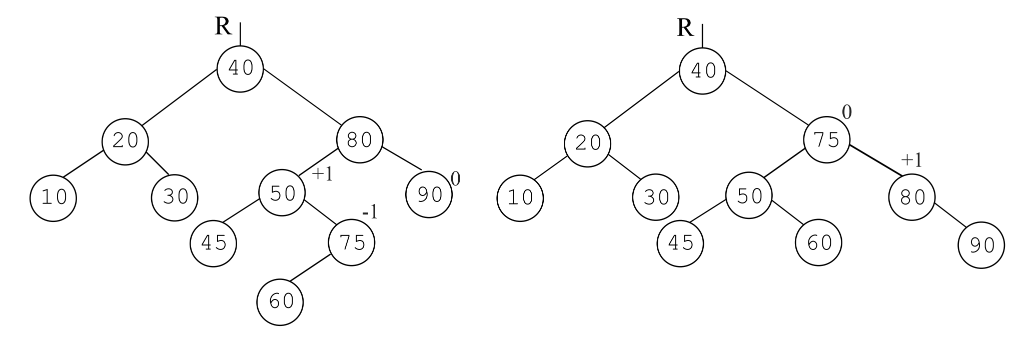


Figura 14.14 *a)*Árbol después de insertar clave `60`; *b)* rotación doble, izquierda derecha

> ***A recordar***
>
> La rotación doble resuelve dos casos simétricos, se pueden denominar rotación `ID` y rotación `DI`. En la rotación doble hay que mover los enlaces de tres nodos, el nodo *padre*, el *descendiente* y el *descendiente del descendiente* por la rama contraria.

14.3.5. Movimiento de enlaces en la rotación doble

Los cambios descritos en la rotación doble afectan a tres nodos el nodo problema `n`, el descendiente por la rama desequilibrada `n1`, y el descendiente de `n1` (por la izquierda o la derecha, según el tipo de rotación doble) apuntado por `n2`. En los dos casos simétricos de rotación doble, rotación *izquierda-derecha* (rotación `ID`) y rotación *derecha-izquierda* (rotación `DI`), el nodo `n2` pasa a ser la raíz del nuevo subárbol.

Los movimientos de los enlace para realizar la rotación `ID` son:

```
n1.dcho = n2.izdo
n2.izdo = n1
```

```
n.izdo  = n2.dcho
n2.dcho = n
n       = n2
```

Los factores de equilibrio de los nodos implicados en la rotación `ID` dependen del factor de equilibrio, antes de la inserción, del nodo apuntado por `n2`, según esta tabla:

Si	n2.fe = -1	n2.fe = 0	n2.fe = 1
	n.fe = 1	0	0
	n1.fe = 0	0	-1
	n2.fe = 0	0	0

Los movimientos de los enlaces para realizar la rotación `DI` (observar la simetría en los movimientos de los enlaces) son:

```
n1.izdo = n2.dcho
n2.dcho = n1
n.dcho  = n2.izdo
n2.izdo = n
n       = n2
```

Los factores de equilibrio de los nodos implicados en la rotación `DI` también dependen del factor de equilibrio del nodo `n2`, según la tabla:

Si	n2.fe = -1	n2.fe = 0	n2.fe = 1
	n.fe = 0	0	-1
	n1.fe = 1	0	0
	n2.fe = 0	0	0

Nota de ejecución

La complejidad del algoritmo de inserción de una clave en un árbol de búsqueda AVL es la suma de la complejidad para *bajar* al nivel de las hojas `(O(log n))` más la complejidad en el peor de los casos de la vuelta por el camino de búsqueda, para actualizar el factor de equilibrio de los nodos que es `O(log n)`, más la complejidad de los movimientos de los enlaces en la rotación, que tiene complejidad constante. En definitiva, la complejidad de la inserción es `O(log n)`, *complejidad logarítmica.*

14.4. IMPLEMENTACIÓN DE LA *INSERCIÓN CON BALANCEO Y ROTACIONES*

La realización de la fase de inserción es igual que la descrita para los árboles de búsqueda. Ahora se añade la fase de actualización de los factores de equilibrio; una vez insertado, se activa un *flag* (indicador) para indicar que ha crecido en altura, de tal forma que al *regresar* por el *camino de búsqueda* calcule los nuevos factores de equilibrio de los nodos que forman el camino. Cuando la inserción se ha realizado por la rama izquierda del nodo, la altura crece por la izquierda y, por

tanto, disminuye en 1 el factor de equilibrio; si se hace por la rama derecha, el factor de equilibrio aumenta en 1. El proceso termina si la altura del subárbol no aumenta. También termina si se produce un desequilibrio, ya que cualquier rotación tiene la propiedad de que la altura del subárbol resultante es la misma que antes de la inserción.

A continuación se escriben los métodos, miembros de la clase ArbolAvl, que implementan los cuatro tipos de rotaciones y de la operación de inserción. Todos devuelven la referencia al nodo raíz del subárbol implicado en la rotación.

Rotaciones

```
private NodoAvl rotacionII(NodoAvl n, NodoAvl n1)
{
   n.ramaIzdo(n1.subarbolDcho());
   n1.ramaDcho(n);
   // actualización de los factores de equilibrio
   if (n1.fe == -1)    // se cumple en la inserción
   {
      n.fe = 0;
      n1.fe = 0;
   }
   else
   {
      n.fe = -1;
      n1.fe = 1;
   }
   return n1;
}

private NodoAvl rotacionDD(NodoAvl n, NodoAvl n1)
{
   n.ramaDcho(n1.subarbolIzdo());
   n1.ramaIzdo(n);
   // actualización de los factores de equilibrio
   if (n1.fe == +1)    // se cumple en la inserción
   {
      n.fe = 0;
      n1.fe = 0;
   }
   else
   {
      n.fe = +1;
      n1.fe = -1;
   }
   return n1;
}

private NodoAvl rotacionID(NodoAvl n, NodoAvl n1)
{
   NodoAvl n2;

   n2 = (NodoAvl) n1.subarbolDcho();
   n.ramaIzdo(n2.subarbolDcho());
   n2.ramaDcho(n);
   n1.ramaDcho(n2.subarbolIzdo());
```

```
      n2.ramaIzdo(n1);
      // actualización de los factores de equilibrio
      if (n2.fe == +1)
         n1.fe = -1;
      else
         n1.fe = 0;
      if (n2.fe == -1)
         n.fe = 1;
      else
         n.fe = 0;
      n2.fe = 0;
      return n2;
   }

   private NodoAvl rotacionDI(NodoAvl n, NodoAvl n1)
   {
      NodoAvl n2;

      n2 = (NodoAvl)n1.subarbolIzdo();

      n.ramaDcho(n2.subarbolIzdo());
      n2.ramaIzdo(n);
      n1.ramaIzdo(n2.subarbolDcho());
      n2.ramaDcho(n1);
         // actualización de los factores de equilibrio
      if (n2.fe == +1)
      n.fe = -1;
      else
      n.fe = 0;

      if (n2.fe == -1)
         n1.fe = 1;
      else
         n1.fe = 0;
      n2.fe = 0;
      return n2;
   }
```

En estos métodos hay actualizaciones del factor de equilibrio que suceden únicamente en el borrado de una clave y se estudian en el siguiente apartado.

Inserción con balanceo

El método `insertar()` es la interfaz de la operación, llama al método interno, recursivo, que realiza la operación y devuelve la raíz del nuevo árbol.

```
   public void insertar (Object valor)throws Exception
   {
      Comparador dato;
      Logical h = new Logical(false); // intercambia un valor booleano
      dato = (Comparador) valor;
      raiz = insertarAvl(raiz, dato, h);
   }

   private NodoAvl
```

```
insertarAvl(NodoAvl raiz, Comparador dt, Logical h) throws Exception
{
   NodoAvl n1;

   if (raiz == null)
   {
      raiz = new NodoAvl(dt);
      h.setLogical(true);

   }
   else if (dt.menorQue(raiz.valorNodo()))
   {
      NodoAvl iz;
      iz = insertarAvl((NodoAvl) raiz.subarbolIzdo(), dt, h);
      raiz.ramaIzdo(iz);
      // regreso por los nodos del camino de búsqueda
      if (h.booleanValue())
      {
         // decrementa el fe por aumentar la altura de rama izquierda
         switch (raiz.fe)
         {
            case  1:
               raiz.fe = 0;
               h.setLogical(false);
               break;
            case  0:
               raiz.fe = -1;
               break;
            case -1:// aplicar rotación a la izquierda
               n1 = (NodoAvl)raiz.subarbolIzdo();
               if (n1.fe == -1)
                  raiz = rotacionII(raiz, n1);
               else
                  raiz = rotacionID(raiz, n1);
               h.setLogical(false);

         }
      }
   }
   else if (dt.mayorQue(raiz.valorNodo()))
   {
      NodoAvl dr;
      dr = insertarAvl((NodoAvl)raiz.subarbolDcho(), dt, h);
      raiz.ramaDcho(dr);
      // regreso por los nodos del camino de búsqueda
      if (h.booleanValue())
      {
         // incrementa el fe por aumentar la altura de rama izquierda
         switch (raiz.fe)
         {
            case  1:   // aplicar rotación a la derecha
               n1 = (NodoAvl)raiz.subarbolDcho();
               if (n1.fe == +1)
                  raiz = rotacionDD(raiz, n1);
```

```
            else
               raiz = rotacionDI(raiz,n1);
            h.setLogical(false);
            break;
         case  0:
            raiz.fe = +1;
            break;
         case -1:
            raiz.fe = 0;
               h.setLogical(false);
       }
    }
  }
  else
     throw new Exception("No puede haber claves repetidas " );
  return raiz;
}
```

Con el fin de transmitir el valor `boolean` entre llamadas al método recursivo `insertarAvl` , se ha utilizado la clase `Logical` que tiene la característica de poder cambiar el valor almacenado (método `setLogical()`). La declaración de esta clase es:

```
package arbolEqui;

public class Logical
{
   boolean v;
   public Logical (boolean f)
   {
      v = f;
   }
   public void setLogical(boolean f)
   {
      v = f;
   }
   public boolean booleanValue()
   {
      return v;
   }
}
```

Ejercicio 14.2

Se desea formar un árbol binario de búsqueda equilibrado de altura 5. El campo dato de cada nodo debe ser una referencia a un objeto que guarda un número entero que será la clave de búsqueda. Una vez formado el árbol, mostrar las claves en orden creciente y el número de nodos de que consta el árbol.

Se declara la clase `Numero` que implementa la interfaz `Comparador` y guarda un valor de tipo `int`. La formación del árbol se hace con repetidas llamadas al método `insertar()`, que pasa control a `insertarAvl()`. La condición para terminar la formación del árbol está expuesta en el enunciado: altura del árbol igual a `5`. El método `altura()` de la clase principal determina dicho

parámetro. También en la clase *principal,* se escribe el método `visualizar()`, basado en el recorrido *inorden*, para mostrar las claves en orden creciente y, a la vez, contar los nodos visitados.

```
import java.io.*;
import arbolBinarioOrdenado.Comparador;
import arbolAvl.*;

class Numero implements Comparador
{
   int valor;
   public Numero(int n)
   {
      valor = n;
   }
   public String toString()
   {
      return " " + valor;
   }
   public boolean menorQue(Object op2)
   {
      Numero p2 = (Numero) op2;
      return  valor < p2.valor;
   }
      // ... de forma similar el resto de operaciones

}

public class ArbolAvlNumerico // clase principal
{
   static final int TOPEMAX = 999;
   public static void main(String [] a) throws Exception
   {
      ArbolAvl avl = new ArbolAvl();
      Numero elemento;
      int numNodos;

      while (altura(avl.raizArbol()) < 5)
      {
         elemento = new Numero((int)(Math.random()*TOPEMAX) + 1);
         avl.insertar(elemento);
      }

      numNodos = visualizar(avl.raizArbol());
      System.out.println("\n Número de nodos: " + numNodos);
   }

   static int visualizar (NodoAvl r) //escribir claves de árbol
   {
      if (r != null)
      {
         int cuantosIzquierda, cuantosDerecha;

         cuantosIzquierda = visualizar((NodoAvl)r.subarbolIzdo());
         System.out.print(r.valorNodo());
         cuantosDerecha = visualizar((NodoAvl)r.subarbolDcho());
```

```
            return cuantosIzquierda + cuantosDerecha + 1;
         }
         else
            return 0;
      }

      static int altura(NodoAvl r) // calcula y devuelve altura
      {
         if (r != null)
            return mayor(altura((NodoAvl)r.subarbolIzdo()),
                         altura((NodoAvl)r.subarbolDcho())) + 1;
         else
            return 0;
      }
      static int mayor (int x, int y)
      {
         return (x > y ? x : y);
      }
   }
```

14.5. BORRADO DE UN NODO EN UN ÁRBOL EQUILIBRADO

Esta operación elimina un nodo, con cierta clave, de un árbol de búsqueda equilibrado; el árbol resultante debe de seguir siendo un árbol equilibrado ($|h_{Ri} - h_{Rd}| <= 1$).

El algoritmo de borrado puede descomponerse en dos partes diferenciadas. La primera sigue la estrategia del borrado en árboles de búsqueda. La segunda consiste en actualizar el *factor de equilibrio,* para lo que recorre el *camino de búsqueda* hacia la raíz, actualizando el factor de equilibrio de los nodos. Ahora, al eliminarse un nodo, la altura de la rama en que se encuentra disminuye. Si después de la actualización de un nodo ocurre que se *viola* la condición de equilibrio, `fe = ± 2`, hay que restaurar del equilibrio con una rotación simple o doble. La Figura 14.15a muestra un árbol equilibrado con el factor de equilibrio de cada nodo; en la Figura 14.15b el árbol, resultante después de eliminar el nodo `15`, y actualizar el factor de equilibrio del nodo `28`. Se observa que el factor de equilibrio del nodo `28` pasa a `+2` ya que se ha eliminado por su rama izquierda, por esa razón hay que establecer el equilibrio en una rotación doble.

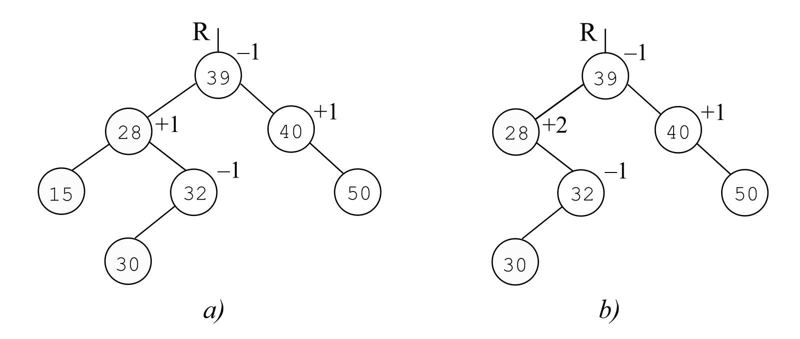


Figura 14.15 *a)* Árbol de búsqueda equilibrado;
b) el árbol después de eliminar el nodo `14`

14.5.1. Algoritmo de borrado

En el algoritmo lo primero que se hace es buscar el nodo con la clave a eliminar, para ello se sigue el *camino de búsqueda*. A continuación, se procede a eliminar el nodo. Se distinguen dos casos:

1. El nodo a borrar es un nodo hoja, o con un único descendiente. Entonces, simplemente se suprime, o bien se sustituye por su descendiente.
2. El nodo a eliminar tiene dos subárboles. Se procede a buscar el nodo más a la derecha del subárbol izquierdo, es decir, el de *mayor clave en el subárbol de claves menores*; éste se copia en el nodo a eliminar y, por último, se retira el nodo copiado (que es una hoja).

Una vez eliminado el nodo, el algoritmo tiene que prever la actualización de los factores de equilibrio de los nodos que han formado el *camino de búsqueda,* ya que la altura de alguna de las dos ramas ha disminuido. Por consiguiente, se *regresa* por los nodos del camino, hacia la raíz, calculando el factor de equilibrio. No siempre es necesario recorrer todo el camino de regreso, si se actualiza un nodo con factor de equilibrio `0`, éste pasará a ser `± 1`, pero la altura neta del subárbol con raíz el nodo actualizado no ha cambiado, y entonces el algoritmo termina ya que los factores de equilibrio de los nodos restantes no cambian. La Figura 14.16*a* muestra un árbol equilibrado, en la Figura 14.16*b* el árbol resultante después de eliminar el nodo con la clave `90`; al volver por el camino de búsqueda, el factor de equilibrio del nodo `70` pasa a ser `0`, el del nodo `60`, a `–1` (ha disminuido su rama derecha). Sin embargo su altura neta no ha cambiado, sigue siendo `3`; por ello, el algoritmo termina.

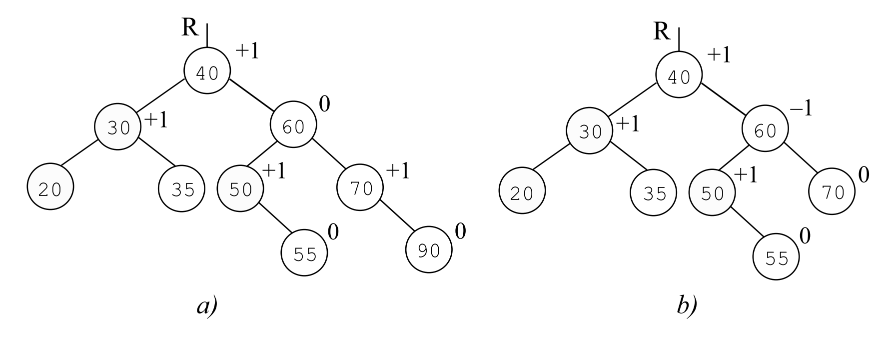


Figura 14.16 *a)* Árbol de búsqueda equilibrado; *b)* el árbol después de eliminar el nodo 90

Otro caso a considerar, en la actualización, es cuando un nodo tenga como `fe = ± 1` y la eliminación se realice por la rama más alta; entonces el equilibrio mejora, pasa a ser cero. Ahora bien, en este caso la altura del subárbol ha disminuido y el algoritmo debe seguir retrocediendo por la nodos del camino, por si es necesario una reestructuración. En la Figura 14.17*b* se tiene el árbol de búsqueda después de eliminar el nodo con la clave `25` del árbol 14.17*a*. Al actualizar los nodos del camino de búsqueda, el nodo `20` tiene como `fe = +1`, éste pasa a ser `0` ya que se ha eliminado por la rama derecha. El proceso continúa ya que la altura del subárbol de raíz `20` ha disminuido, el nodo antecesor, `29`, tiene como `fe = +1`, pasa a `+2` ya que se ha eliminado por su rama izquierda, *viola* la condición de equilibrio. Para restaurar el equilibrio es necesario una rotación simple, *derecha-derecha,* ya que el nodo de la rama derecha de `29` tiene `fe = 0`. La Figura 14.17*c* es el árbol equilibrado después de la rotación.

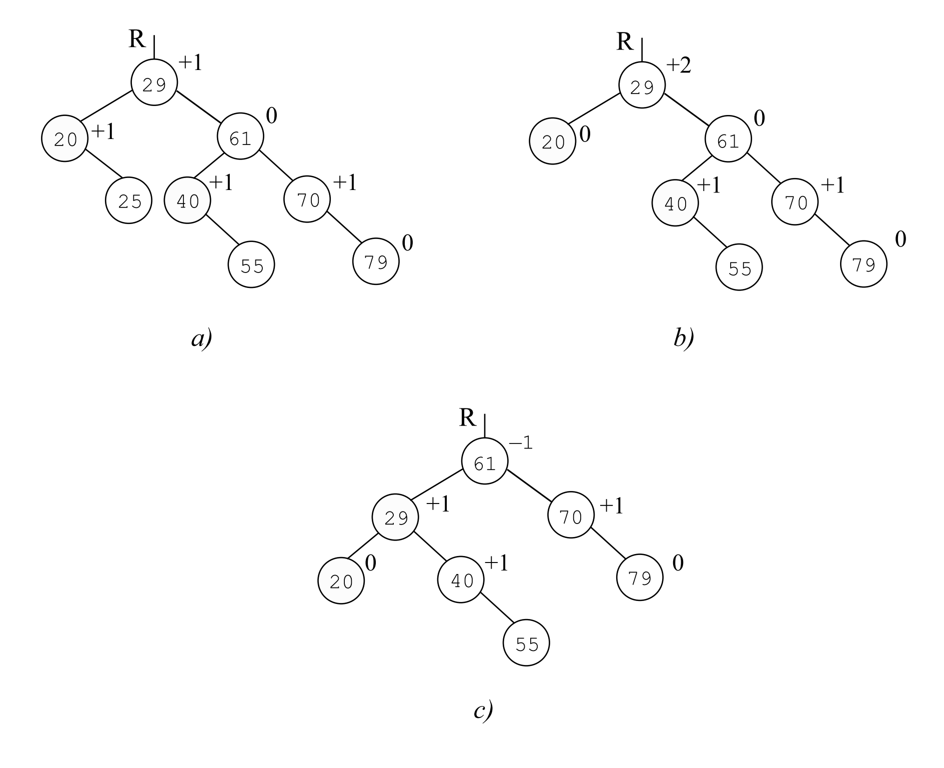


Figura 14.17 *a)* Árbol de búsqueda equilibrado; *b)* el árbol después de eliminar el nodo `25`; *c)* árbol una vez realizada rotación *derecha-derecha*

Aquellos nodos que tienen como `fe = ± 1`, y al actualizarlos su factor de equilibrio vale `± 2` hay que reestructurarlos, con una rotación simple o doble. El tipo específico de rotación depende del `fe` del nodo problema, apuntado por `n`, y del nodo descendiente `n1`:

- Si `n.fe == +2`, entonces `n1` es su *hijo* derecho, de tal forma que si `n1.fe ≥ 0` la rotación a aplicar es *derecha-derecha*. Y si `n1.fe == -1`, la rotación a aplicar es *derecha-izquierda*.
- De forma simétrica, si `n.fe == -2`, entonces `n1` es su *hijo* izquierdo, de tal forma que si `n1.fe ≤ 0` la rotación a aplicar es *izquierda-izquierda*. Y si `n1.fe == +1`, la rotación a aplicar es *izquierda-derecha*.

En el proceso de eliminar una clave una vez que se aplica una rotación, la altura del subárbol puede disminuir, por ello el proceso de actualización del `fe` debe continuar, ya que otro nodo del camino hacia la raíz puede que viole la condición de equilibrio y sea necesario aplicar otra rotación. La Figura 14.18*b* es un árbol de búsqueda después de haber eliminado la clave `21`; con la actualización del factor de equilibrio, el nodo `43` está desequilibrado, hay que aplicar una rotación *derecha-derecha*. La Figura 14.18*c* muestra el árbol después de la rotación y la posterior actualización del factor de equilibrio del nodo `70`, que también exige aplicar otra rotación, en este caso *derecha-izquierda*. La Figura 14.18*d* es el árbol equilibrado después de la última rotación y la finalización del algoritmo.

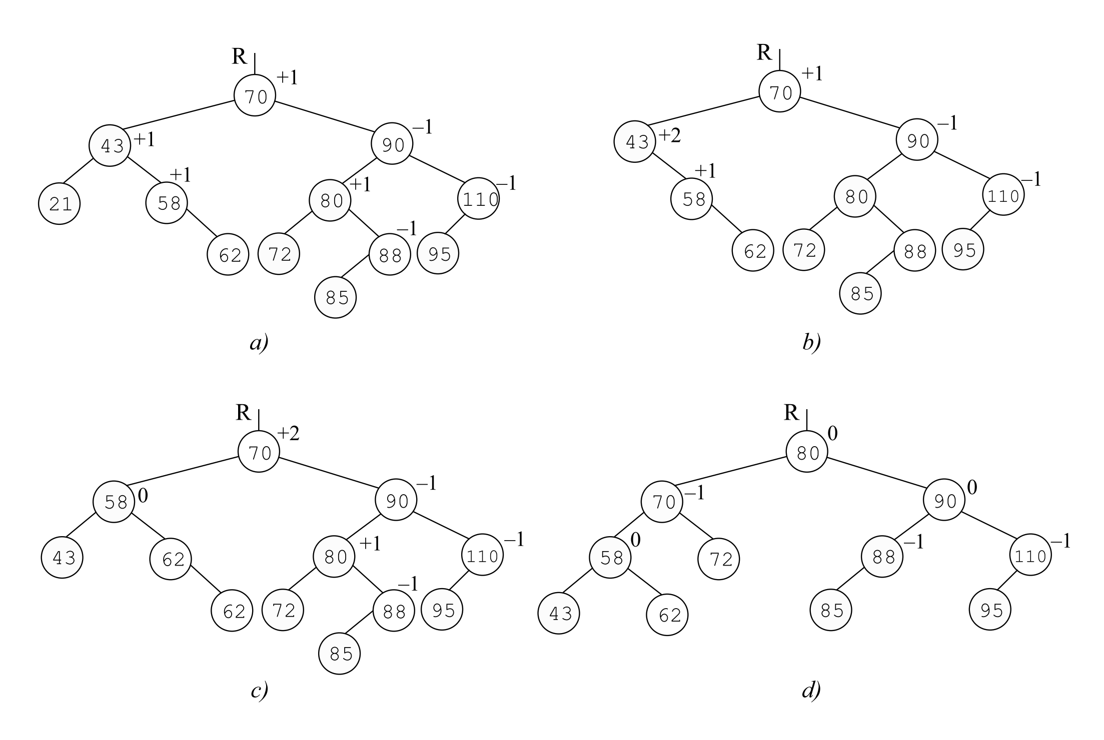


Figura 14.18 *a)* Árbol de búsqueda equilibrado; *b)* el árbol después de eliminar el nodo `21`; *c)* árbol una vez realizada la rotación *derecha-derecha*; *d)* árbol después de la nueva rotación, *derecha-izquierda*

> ***Nota***
>
> El algoritmo de borrado de una clave en un árbol de búsqueda AVL puede necesitar aplicar más de una rotación para que el árbol resultante siga siendo equilibrado. En cuanto a la complejidad, es la suma de la complejidad para encontrar el nodo, `O(log n)`, más la complejidad de la *vuelta por el camino de búsqueda,* `O(log n)`, más la complejidad de los movimientos de los enlaces en la rotación o rotaciones, que es una complejidad constante. En definitiva, la complejidad de la operación es *logarítmica*.

14.5.2. Implementación de la operación borrado

El método que implementa la operación utiliza el argumento `cambiaAltura` para indicar que se ha producido un cambio en la altura de una rama. Éste se activa cuando la altura del subárbol disminuye, debido a la eliminación de un nodo, o bien por la aplicación de una rotación. Además, con el fin de diferenciar los diferentes casos que ocurren cuando disminuye la altura, se introducen dos métodos internos, `equilibrar1()` y `equilibrar2()`. El primero se invoca cuando la altura de la rama izquierda disminuye y, de forma simétrica, cuando disminuye la altura de la rama derecha se invoca `equilibrar2()`.

En `equilibrar1()`, al disminuir la altura de la rama izquierda, el factor de equilibrio se incrementa; en caso de *violarse* el equilibrio se aplica la rotación del tipo *derecha-derecha* o

derecha-izquierda. En `equilibrar2()`, al disminuir la altura de la rama derecha, el factor de equilibrio disminuye, por ello de *violarse* el equilibrio se aplica la rotación del tipo *izquierda-izquierda* o *izquierda-derecha*.

Los métodos que implementan las rotaciones se han descrito en el apartado14.4. Hay cambios, en los factores de equilibrio, que sólo pueden ocurrir en la operación de borrado.

La interfaz de la operación es el método `eliminar()`, que pasa control a `borrarAvl()`. Este método implementa el algoritmo de borrado; para ello describe un *camino de búsqueda* hasta encontrar el nodo con el elemento a eliminar según la rama en que se encuentre, llama a `equilibrar1()` o `equilibrar2()`.

```
public void eliminar (Object valor) throws Exception
{
   Comparador dato;
   dato = (Comparador) valor;
   Logical flag = new Logical(false);
   raiz = borrarAvl(raiz, dato, flag);
}

private NodoAvl borrarAvl(NodoAvl r, Comparador clave,
                          Logical cambiaAltura) throws Exception
{
   if (r == null)
   {
      throw new Exception (" Nodo no encontrado ");
   }
   else if (clave.menorQue(r.valorNodo()))
   {
      NodoAvl iz;
      iz = borrarAvl((NodoAvl)r.subarbolIzdo(), clave, cambiaAltura);
      r.ramaIzdo(iz);
      if (cambiaAltura.booleanValue())
         r = equilibrar1(r, cambiaAltura);
   }
   else if (clave.mayorQue(r.valorNodo()))
   {
      NodoAvl dr;
      dr = borrarAvl((NodoAvl)r.subarbolDcho(), clave, cambiaAltura);
      r.ramaDcho(dr);
      if (cambiaAltura.booleanValue())
         r = equilibrar2(r, cambiaAltura);
   }
   else // Nodo encontrado
   {
      NodoAvl q;
      q = r;   // nodo a quitar del árbol
      if (q.subarbolIzdo()== null)
      {
         r = (NodoAvl) q.subarbolDcho();
         cambiaAltura.setLogical(true);
      }
      else if (q.subarbolDcho() == null)
      {
         r = (NodoAvl) q.subarbolIzdo();
         cambiaAltura.setLogical(true);
      }
      else
      {       // tiene rama izquierda y derecha
```

```
         NodoAvl iz;
         iz = reemplazar(r, (NodoAvl)r.subarbolIzdo(), cambiaAltura);
         r.ramaIzdo(iz);
         if (cambiaAltura.booleanValue())
            r = equilibrar1(r, cambiaAltura);
      }
      q = null;
   }
   return r;
}

private
NodoAvl reemplazar(NodoAvl n, NodoAvl act, Logical cambiaAltura)
{
   if (act.subarbolDcho() != null)
   {
      NodoAvl d;
      d = reemplazar(n, (NodoAvl)act.subarbolDcho(), cambiaAltura);
      act.ramaDcho(d);
      if (cambiaAltura.booleanValue())
         act = equilibrar2(act, cambiaAltura);
   }
   else
   {
      n. nuevoValor(act.valorNodo());
      n = act;
      act = (NodoAvl)act.subarbolIzdo();
      n = null;
      cambiaAltura.setLogical(true);
   }
   return act;
}

private NodoAvl equilibrar1(NodoAvl n, Logical cambiaAltura)
{
   NodoAvl n1;

   switch (n.fe)
   {
      case -1 :  n.fe = 0;
                 break;
      case 0  :  n.fe = 1;
                 cambiaAltura.setLogical(false);
                 break;
      case +1 :  //se aplicar un tipo de rotación derecha
                 n1 = (NodoAvl)n.subarbolDcho();
                 if (n1.fe >= 0)
                 {
                    if (n1.fe == 0)  //la altura no vuelve a disminuir
                         cambiaAltura.setLogical(false);
                    n = rotacionDD(n, n1);
                 }
                 else
                    n = rotacionDI(n, n1);
                 break;
   }
   return n;
}
private NodoAvl equilibrar2(NodoAvl n, Logical cambiaAltura)
```

```
{
   NodoAvl n1;

   switch (n.fe)
   {
      case -1:// Se aplica un tipo de rotación izquierda
               n1 = (NodoAvl)n.subarbolIzdo();
               if (n1.fe <= 0)
               {
                  if (n1.fe == 0)
                      cambiaAltura.setLogical(false);
                  n = rotacionII(n, n1);
               }
               else
                  n = rotacionID(n,n1);
               break;
      case 0  :  n.fe = -1;
               cambiaAltura.setLogical(false);
               break;
      case +1 :  n.fe = 0;
               break;
    }

      return n;
}
```

RESUMEN

Los árboles binarios de búsqueda son estructuras de datos importantes que permiten localizar una *clave de búsqueda* con un coste logarítmico. Sin embargo, el tiempo de búsqueda puede crecer hasta hacerse lineal si el árbol está degenerado; en ese sentido, la eficiencia de las operaciones depende de lo aleatorias que sean las claves de entrada. Para evitar este problema, surgen los árboles de búsqueda equilibrados, llamados árboles AVL. La eficiencia de la *búsqueda* en los árboles equilibrados es mayor que en los árboles de búsqueda no equilibrados.

Un árbol binario equilibrado es un árbol de búsqueda caracterizado por diferir las alturas de las ramas derecha e izquierda del nodo raíz a lo sumo en uno y por que los subárboles izquierdo y derecho son, a su vez, árboles equilibrados.

Las operaciones de los árboles de búsqueda son también operaciones de los árboles equilibrado. Las operaciones de inserción y de borrado añaden y eliminan, respectivamente, un nodo en el árbol de igual forma que en los árboles de búsqueda no equilibrados. Además, es necesario incorporar una segunda parte de reestructuración del árbol para asegurar el equilibrio de cada nodo. La solución que se aplica ante un desequilibrio de un nodo viene dada por las *rotaciones*. Se pueden aplican dos tipos de rotaciones, *rotación simple y rotación doble*; el que se aplique una u otra depende del nodo desequilibrado y del *equilibrio* del hijo que se encuentre en la rama más alta.

Las operaciones de inserción y de borrado en árboles binarios equilibrados son más *costosas* que en los no equilibrados, debido a que el algoritmo debe *retroceder* por el *camino de búsqueda* para actualizar el factor de equilibrio de los nodos que lo forman; a esta acción se suma el *coste* de la rotación cuando se viola la condición de equilibrio. Por el contrario, la operación de búsqueda es, de promedio, menos costosa que en los árboles binarios no equilibrados.

EJERCICIOS

14.1. Dibujar el árbol binario de búsqueda equilibrado que se produce con las claves: 14, 6, 24, 35, 59, 17, 21, 32, 4, 7, 15 y 22.

14.2. Dada la secuencia de claves enteras: 100, 29, 71, 82, 48, 39, 101, 22, 46, 17, 3, 20, 25, 10, dibujar el árbol AVL correspondiente. Eliminar claves consecutivamente hasta encontrar un nodo que *viole* la condición de equilibrio y cuya restauración sea con una rotación doble.

14.3. En el árbol construido en el Ejercicio 14.1 eliminar el nodo raíz. Hacerlo tantas veces como sea necesario hasta que se desequilibre un nodo y haya que aplicar una rotación simple.

14.4. Encontrar una secuencia de *n* claves que al ser insertadas en un árbol binario de búsqueda vacío permiten aplicar las cuatro rutinas de rotación: `II, ID, DD, DI`.

14.5. Dibujar el árbol equilibrado después de insertar en orden creciente 31 ($2^5 - 1$) elementos del 11 al 46.

14.6. ¿Cuál es el número mínimo de nodos de un árbol binario de búsqueda equilibrado de altura 10?

14.7. Escribir el método recursivo `buscarMin()` de forma que devuelva el nodo de clave mínima de un árbol de búsqueda equilibrado.

14.8. Dibujar un árbol AVL de altura 6 con el criterio del *peor de los casos*; es decir, aquel en el que cada nodo tenga como factor de equilibrio ± 1.

14.9. En el árbol equilibrado formado en el Ejercicio 14.8, eliminar una de las hojas menos profundas. Representar las operaciones necesarias para restablecer el equilibrio.

14.10. Escribir el método recursivo `buscarMax()` que devuelva el nodo de clave máxima de un árbol de búsqueda equilibrado.

14.11. Escribir los métodos `buscarMin()` y `buscarMax()` en un árbol de búsqueda equilibrado de manera iterativa.

PROBLEMAS

14.1. Dado un archivo de texto, construir un árbol AVL con todas sus palabras y frecuencia. El archivo se denomina *carta.dat*.

14.2. Añadir al programa escrito en el Problema 14.1 un método que, dada una palabra, devuelva el número de veces que aparece en el texto.

14.3. En el archivo *alumnos.txt* se encuentran los nombres completos de los alumnos de las escuelas taller de la Comunidad Alcarreña. Escribir un programa para leer el archivo y formar, inicialmente, un árbol de búsqueda con respecto a la clave *apellido*. Una vez formado el árbol, construir con sus nodos un árbol de Fibonacci.

14.4. La implementación de la operación *insertar* en un árbol equilibrado se realiza de manera natural en forma recursiva. Escribir de nuevo la codificación aplicando una estrategia iterativa.

14.5. Un archivo F contiene las claves, enteros positivos, que forman un árbol binario de búsqueda equilibrado R. El archivo se grabó en el recorrido por niveles del árbol R. Escribir un programa que realice las siguientes tareas: a) Leer el archivo F para volver a construir el árbol equilibrado. b) Buscar una clave, requerida al usuario, y en el caso de que esté en el árbol mostrar las claves que se encuentran en el mismo nivel del árbol.

14.6. La implementación de la operación *eliminar* en un árbol binario equilibrado se ha realizado en forma recursiva. Escribir de nuevo la codificación aplicando una estrategia iterativa.

14.7. En un archivo se ha almacenado los habitantes de *n* pueblos de la comarca natural *Peñas Rubias*. Cada registro del archivo tiene el nombre del pueblo y el número de habitantes. Se desea asociar los nombres de cada habitante a cada pueblo, para ello se ha pensado en una estructura de datos que consta de un *array* de n elementos. Cada elemento tiene el nombre del pueblo y la raíz de un árbol AVL con los nombres de los habitantes del pueblo.

Escribir un programa que cree la estructura. Como entrada de datos, utilizar los nombres de los habitantes que se insertarán en el árbol AVL del pueblo que le corresponde.

14.8. La operación de borrar una clave en un árbol AVL se ha realizado de tal forma que cuando el nodo de la clave a eliminar tiene las dos ramas, se reemplaza por la clave mayor del subárbol izquierdo. Implementar la operación de borrado de tal forma que cuando el nodo a eliminar tenga dos ramas, se reemplace, aleatoriamente, por la clave mayor de la rama izquierda o por la clave menor de la rama derecha.

14.9. Al Problema 14.7 se desea añadir la posibilidad de realizar operaciones sobre la estructura. Así, añadir la posibilidad de cambiar el nombre de una persona de un determinado pueblo. Esta operación debe mantener el árbol como árbol de búsqueda, ya que al cambiar el nombre y ser la clave de búsqueda el nombre puede ocurrir que se rompa la condición. Otra opción que debe permitir es dar de baja un pueblo entero de tal forma que todos sus habitantes se añadan a otro pueblo de la estructura. Por último, una vez que se vaya a terminar la ejecución del programa, grabar en un archivo cada pueblo con sus respectivos habitantes.

14.10. Una empresa de servicios tiene tres departamentos: *comercial(1)*, *explotación(2)* y *marketing(3)*. Cada empleado está adscrito a uno de ellos. Se ha realizado una redistribución del personal entre ambos departamentos, los cambios están guardados en el archivo *laboral.txt*. El archivo contiene en cada registro los campos *identificador, origen, destino*. El campo *origen* puede tomar los valores 1, 2, 3 dependiendo del departamento origen del empleado, cuya identificación es una secuencia de 5 dígitos que es el primer campo del registro. El campo *destino* también puede tomar los valores 1, 2, 3 según el departamento al que sea destinado.

Escribir un programa que guarde los registros del archivo en tres árboles AVL, uno por cada departamento origen, y realice los intercambios de registros en los árboles según el campo *destino*.



CAPITULO 15

Grafos, representación y operaciones

Objetivos

Con el estudio de este capítulo, usted podrá:

- Distinguir entre relaciones jerárquicas y otras relaciones.
- Definir un grafo e identificar sus componentes.
- Conocer estructuras de datos para representar un grafo.
- Conocer las operaciones básicas que se aplican sobre grafos.
- Encontrar los caminos que puede haber entre dos nodos de un grafo.
- Realizar en Java la representación de los grafos y las operaciones básicas.

Contenido

Conceptos clave

- Camino.
- Conexión y componente conexa.
- Factor de peso.
- Lista de adyacencia.
- Matriz de adyacencia.
- Relación jerárquica.
- Vértice y arco.

INTRODUCCIÓN

Este capítulo introduce al lector a conceptos matemáticos importantes denominados *grafos* que tienen aplicaciones en campos tan diversos como sociología, química, geografía, ingeniería eléctrica e industrial, etc. Los grafos se estudian como estructuras de datos o tipos abstractos de datos. Este capítulo estudia las definiciones relativas a los grafos y la representación de los grafos en la memoria del ordenador. El capítulo investiga dos formas tradicionales de implementación de grafos, matriz de adyacencia y listas de adyacencia. También se estudian operaciones importantes y algoritmos de grafos que son significativos en informática.

15.1. CONCEPTOS Y DEFINICIONES

Un grafo G agrupa *entes* físicos o conceptuales y las relaciones entre ellos. Un grafo está formado por un conjunto de vértices o nodos `V`, que representan a los *entes,* y un conjunto de arcos `A`, que representan las relaciones entre vértices. Se representa con el par `G = (V, A)`. La Figura 15.1 muestra un grafo formado por los vértices `V = {1,4,5,7,9}` y el conjunto de arcos `A = {(1,4), (4,1), (5,1), (1, 5), (7,9), (9,7), (7,5), (5,7), (4,9), (9,4)}`.

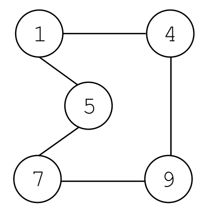


Figura 15.1 Grafo no dirigido

Un arco o arista representa una *relación* entre dos nodos. Esta relación, al estar formada por dos nodos, se representa por `(u, v)` siendo `u, v` el par de nodos. El grafo es *no dirigido* si los arcos están formados por pares de nodos no ordenados, no apuntados; se representa con un segmento uniendo los nodos, `u – v`. El grafo de la Figura 15.1 es no dirigido.

Un grafo es dirigido, también denominado *digrafo,* si los pares de nodos que forman los arcos son ordenados; se representan con una flecha que indica la dirección de la relación, u → v. El grafo de la Figura 15.2, que consta de los vértices `V = {C,D,E,F,H}` y de los arcos `A = {(C,D,), (D,F), (E,H), (H,E), (E,C)}` forma el grafo dirigido `G = {V,A}`

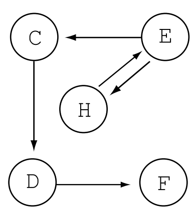


Figura 15.2 Grafo dirigido

Dado el arco `(u,v)` de un grafo, se dice que los vértices `u` y `v` son adyacentes. Si el grafo es dirigido, el vértice `u` es adyacente a `v`, y `v` es adyacente de `u`.

En los modelos realizados con grafos, a veces, una relación entre dos nodos tiene asociada una *magnitud*, denominada factor de peso, en cuyo caso se dice que es un *grafo valorado.* Por ejemplo, los pueblos que forman una comarca junto con la relación *entre un par de pueblos que están unidos por un camino:* esta relación tiene asociado el factor de peso, que es la distancia en kilómetros. La Figura 15.3 muestra un grafo valorado en el que cada arco tiene asociado un peso que es la longitud entre dos nodos.

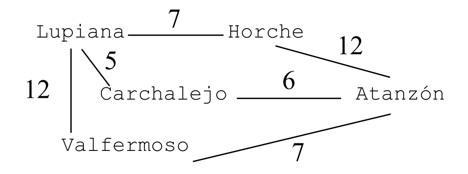


Figura 15.3 Grafo no dirigido valorado

> ***Definición***
>
> Un grafo permite modelar relaciones arbitrarias entre objetos. Un grafo `G = (V,A)` es un par formado por un conjunto de vértices o nodos, `V`, y un conjunto de arcos o aristas, `A`. Cada arco es el par `(u,w)`, siendo `u, w` dos vértices relacionados.

15.1.1. Grado de entrada, grado de salida de un nodo

El *grado* es una cualidad que se refiere a los nodos de un grafo. En un grafo no dirigido, el grado de un nodo `v`, `grado(v)`, es el número de arcos que contienen a `v`. En un grafo dirigido se distingue entre grado de entrada y grado de salida; *grado de entrada* de un nodo `v`, `gradent(v)`, es el número de arcos que llegan a `v`; *grado de salida* de `v`, `gradsal(v)`, es el número de arcos que salen de `v`.

Así, en el grafo no dirigido de la Figura 15.3, `grado(Lupiana) = 3`. En el grafo dirigido de la Figura 15.2, `gradent(D) = 1` y `gradsal(D) = 1`.

15.1.2. Camino

Un camino `P` de longitud `n` desde el vértice v_0 a v_n en un grafo `G`, es la secuencia de `n+1` vértices $v_0, v_1, v_2, \ldots, v_n$ tal que $(v_i, v_{i+1}) \in A$ *(arcos)* para $0 \leq i \leq n$. Matemáticamente, el camino $P = (v_0, v_1, v_2, \ldots, v_n)$.

En el grafo de la Figura 15.4 se pueden encontrar más de un camino; por ejemplo, P_1 `= (4,6,9,7)` es un camino de longitud `3`. P_2 `= (10, 11)` es un camino de longitud `1`. En resumen, la *longitud del camino* es el número de arcos que lo forma.

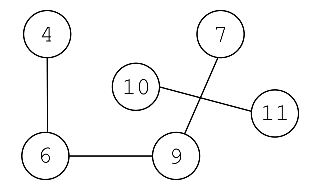


Figura 15.4 Grafo no dirigido de 5 vértices

Definición

La longitud de un camino es el número de arcos del camino. En un grafo valorado, la longitud del camino con pesos es la suma de los pesos de los arcos en el camino.

En el grafo valorado de la Figura 15.3, el camino `(Lupiana, Valfermoso, Atanzón)` tiene de longitud `12 + 7 = 19`.

En algunos grafos se dan arcos desde un vértice a sí mismo, `(v,v)`; entonces, el camino v → v es un bucle. Normalmente, en los grafos no hay nodos relacionados con sí mismo, no es frecuente encontrarse grafos con bucles.

Un camino $P = (v_0, v_1, v_2, \ldots, v_n)$ es simple si todos los nodos que forman el camino son distintos, pudiendo ser iguales v_0, v_n, es decir, los extremos del camino.

En un grafo dirigido, un ciclo es un camino simple cerrado. Por tanto, un ciclo empieza y termina en el mismo nodo, $v_0 = v_n$, y además, debe tener más de un arco. Un grafo dirigido sin ciclos (acíclico) se acostumbra a denominar GDA *(Grafo Dirigido Acíclico)*. La Figura 15.5 muestra un grafo dirigido en el que los vértices `(A, E, B, F, A)` forman un ciclo de longitud `4`. En general, un ciclo de longitud `k` se denomina *k-ciclo*.

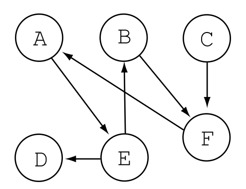


Figura 15.5 Grafo dirigido con ciclos

Un grafo no dirigido es *conexo* si existe un camino entre cualquier par de nodos que forman el grafo. Un grafo dirigido con esta propiedad se dice que es *fuertemente conexo*. Además, un *grafo completo* es aquel que tiene un arco para cualquier par de vértices.

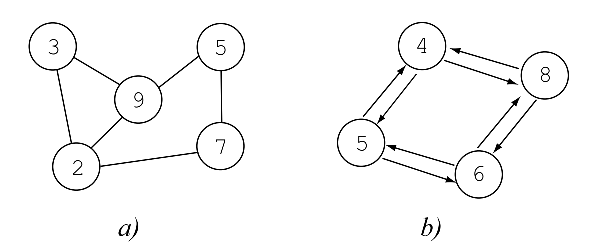


Figura 15.6 *a)* Grafo conexo; *b)* grafo fuertemente conexo

15.1.3. *Tipo Abstracto de Datos* Grafo

Es preciso definir las operaciones básicas para construir la estructura grafo y, en general, modificar sus elementos. En definitiva, especificar el *tipo abstracto de datos grafo*.

Ahora se definen operaciones básicas, a partir de las cuales se construye el grafo. Su realización depende de la representación elegida (matriz de adyacencia, o listas de adyacencia).

arista (u, v).	Añade el arco o arista (u,v) al grafo.
aristaPeso(u,v, w).	Para un grafo valorado, añade el arco (u,v) al grafo y el coste del arco, w.
borraArco(u,v).	Elimina del grafo el arco(u,v).
adyacente(u,v).	Operación que devuelve *cierto* si los vértices u, v forman un arco.
nuevoVértice(u).	Añade el vértice u al grafo G.
borraVértice(u).	Elimina el vértice u del grafo G.

15.2. REPRESENTACIÓN DE LOS GRAFOS

Para trabajar con los grafos y aplicar algoritmos que permitan encontrar propiedades entre los nodos hay que pensar cómo representarlo en memoria interna, qué tipos o estructuras de datos se deben utilizar para considerar los nodos y los arcos.

Una primera simplificación es considerar los vértices o nodos como números consecutivos, empezando por el vértice 0. Es preciso tener en cuenta que se ha de representar un número (finito) de vértices y de arcos que unen dos vértices. Se puede elegir una representación secuencial, mediante un *array* bidimensional, conocida como *matriz de adyacencia*; o bien, una representación dinámica, mediante una estructura multienlazada, denominada *listas de adyacencia*. La elección de una representación u otra depende del tipo de grafo y de las operaciones que se vayan a realizar sobre los vértices y arcos. Para un grafo denso (tiene la mayoría de los arcos posibles) lo mejor es utilizar una matiz de adyacencia. Para un grafo disperso (tiene, relativamente, pocos arcos) se suelen utilizar listas de adyacencia que se ajustan al número de arcos.

15.2.1. Matriz de adyacencia

La característica mas importante de un grafo, que distingue a uno de otro, es el conjunto de pares de vértices que están *relacionados*, o que son adyacentes. Por ello, la forma más sencilla de representación es mediante una matriz, de tantas filas/columnas como nodos, que permite modelar fácilmente esa cualidad.

Sea $G = (V, A)$ un grafo de n nodos, siendo $V = \{v_0, v_1, \ldots, v_{n-1}\}$ el conjunto de nodos, y $A = \{(v_i, v_j)\}$ el conjunto de arcos. Los nodos están numerados consecutivamente de 0 a $n-1$. La representación de los arcos se hace con una matriz A de $n \times n$ elementos, denominada matriz de adyacencia, tal que todo elemento a_{ij} puede tomar los valores:

$$a_{ij} \begin{cases} 1 & \text{si hay un arco } (v_i, v_j) \\ 0 & \text{si no hay arco } (v_i, v_j) \end{cases}$$

Ejemplo 15.1

Dado el grafo dirigido de la Figura 15.7 escribir la matriz de adyacencia.

Suponiendo que el orden de los vértices es {D,F,K,L,R}, entonces la matriz de adyacencia:

$$A = \begin{vmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{vmatrix}$$

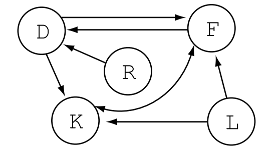
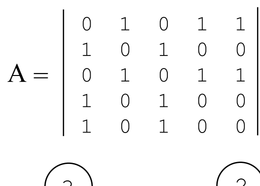


Figura 15.7 Grafo dirigido con los vértices {D,F,K,L,R}

Ejemplo 15.2

Dado el grafo no dirigido de la Figura 15.8 escribir la matriz de adyacencia.

El grafo está formado por 5 vértices. La matriz de adyacencia:



$$A = \begin{vmatrix} 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \end{vmatrix}$$

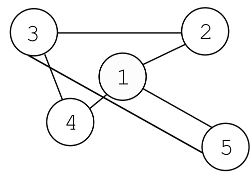


Figura 15.8 Grafo no dirigido con 5 vértices

En los grafos no dirigidos la matriz de adyacencia siempre es simétrica ya que las relaciones entre vértices no son ordenadas: si v_i *está relacionado* con v_j, entonces v_j *está relacionado* con v_i.

Los grafos que modelan problemas en los que un arco tiene asociado una magnitud, un *factor de peso*, también se representan mediante una matriz de tantas filas/columnas como nodos. Ahora un elemento cualquiera, a_{ij} representa el *coste* o *factor de peso* del arco (v_i, v_j). Un arco que no existe se puede representar con un valor imposible, por ejemplo *infinito*; también, se puede representar con el valor 0, dependiendo de que 0 no pueda ser un factor de peso significativo de un arco. A esta matriz también se la denomina *matriz valorada*.

Ejemplo 15.3

Dado el grafo valorado dirigido de la Figura 15.9, escribir la matriz de pesos.

El grafo es un *grafo dirigido con factor de peso*. Si los vértices se numeran en el orden de V = {Alicante, Barcelona, Cartagena, Murcia, Reus}, la matriz de pesos es P y en ella se representa con 0 la no existencia de arco:

$$P = \begin{vmatrix} 0 & 4 & 5 & 0 & 0 \\ 0 & 0 & 3 & 6 & 3 \\ 3 & 0 & 0 & 0 & 0 \\ 5 & 2 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{vmatrix}$$

Figura 15.9 Grafo dirigido con factor de peso

> ***Nota***
>
> La matriz de adyacencia representa los arcos, relaciones entre un par de nodos de un grafo. Es una matriz de unos y ceros, que indican si dos vértices son adyacentes o no. En un grafo valorado, cada elemento representa el peso de la arista, y por ello se la denomina *matriz de pesos*.

15.2.2. Matriz de adyacencia: Clase `GrafoMatriz`

La clase `Vertice` representa un nodo del grafo, con su nombre `(String)` y número asignado. El constructor inicializa el nombre y pone como número de vértice `-1`; el método que añade el vértice al grafo establece el número que le corresponda.

```
package Grafo;

public class Vertice
{
   String nombre;
   int numVertice;
   public Vertice(String x)
   {
      nombre = x;
      numVertice = -1;
   }

   public String nomVertice() // devuelve identificador del vértice
   {
      return nombre;
   }

   public boolean equals(Vertice n) // true, si dos vértices son iguales
   {
      return nombre.equals(n.nombre);
   }

   public void asigVert(int n) // establece el número de vértices
```

```
      {
         numVertice = n;
      }

      public String toString() // características del vértice
      {
         return nombre + " (" + numVertice + ")";
      }
   }
```

La clase `GrafoMatriz` define la matriz de adyacencia, el *array* de vértices y los métodos para añadir nodos y arcos al grafo. La declaración de la clase es:

```
package Grafo;

public class GrafoMatriz
{
   int numVerts;
   static int MaxVerts = 20;
   Vertice [] verts;
   int [][] matAd;
   ...
```

Constructor

El constructor sin argumentos crea la matriz de adyacencia para un máximo de vértices preestablecido; el otro constructor tiene un argumento con el máximo número de vértices:

```
public GrafoMatriz()
{
   this(maxVerts);
}

public GrafoMatriz(int mx)
{
   matAd = new int [mx][mx];
   verts = new Vertice[mx];
   for (int i = 0; i < mx; i++)
      for (int j = 0; i < mx; i++)
         matAd[i][j] = 0;
   numVerts = 0;
}
```

Añadir un vértice

La operación recibe la etiqueta `(String)` de un vértice del grafo, comprueba si ya está en la lista de vértices, en caso negativo incrementa el número de vértices y lo asigna a la lista.

```
public void nuevoVertice (String nom)
{
   boolean esta = numVertice(nom) >= 0;
   if (!esta)
   {
      Vertice v = new Vertice(nom);
      v.asigVert(numVerts);
      verts[numVerts++] = v;
   }
}
```

`numVertice()` busca el vértice en el *array*. Devuelve `-1` si no lo encuentra:

```
boolean int numVertice(String vs)
{
   Vertice v = new Vertice(vs);
   boolean encontrado = false;
   int i = 0;
   for (; (i < numVerts) && !encontrado;)
   {
      encontrado = verts[i].equals(v);
      if (!encontrado) i++ ;
   }
   return (i < numVerts) ? i : -1 ;
}
```

Añadir un arco

El método recibe el nombre de cada vértice del arco, busca, en el *array,* el número de vértice asignado a cada uno de ellos y marca la matriz de adyacencia.

```
public void nuevoArco(String a, String b)throws Exception
{
   int va, vb;
   va = numVertice(a);
   vb = numVertice(b);
   if (va < 0 || vb < 0) throw new Exception ("Vértice no existe");
   matAd[va][vb] = 1;
}
```

Otra versión del método recibe directamente los números de vértice del arco.

```
public void nuevoArco(int va, int vb)throws Exception
{
   if (va < 0 || vb < 0) throw new Exception ("Vértice no existe");
   matAd[va][vb] = 1;
}
```

Para *grafos valorados* este método tiene un tercer argumento que es el *factor de peso* del arco.

> ***Ejecución***
>
> El tiempo de ejecución de la operación que realiza la entrada completa del grafo en memoria depende de la densidad del grafo si se considera un grafo denso el tiempo de ejecución es cuadrático, $O(n^2)$.

Adyacente

Determina si dos vértices, `v1` y `v2`, forman un arco; es decir, si el elemento de la matriz de adyacencia es `1`. Se escriben dos versiones.

```
public boolean adyacente(String a, String b)throws Exception
{
   int va, vb;
   va = numVertice(a);
```

```
      vb = numVertice(b);
      if (va < 0 || vb < 0) throw new Exception ("Vértice no existe");
      return matAd[va][vb] == 1;
   }
   public boolean adyacente(int va, int vb)throws Exception
   {
      if (va < 0 || vb < 0) throw new Exception ("Vértice no existe");
      return matAd[va][vb] == 1;
   }
```

Ejercicio 15.1

Dado un grafo no dirigido de n nodos representado por su matriz de adyacencia, escribir una aplicación que realice su entrada en memoria.

El grafo se guarda en memoria utilizando la clase `GrafoMatriz`. Al constructor de la clase se le pasa el número de nodos del grafo. El usuario sólo tiene que introducir los nombres de los vértices; la llamada al método `nuevoVertice()` crea el vértice y lo asigna a la estructura. A continuación se crean los arcos, para lo cual se leen los pares de vértices con sus nombres y se llama al método `nuevoArco()`.

(Codificación se encuentra en la página web del libro, archivo Ejercicio 15.1.)

15.3. LISTAS DE ADYACENCIA

La representación de un grafo con matriz de adyacencia no es eficiente cuando el grafo es poco denso (disperso), es decir, tiene pocos arcos, y por tanto la matriz de adyacencia tiene muchos ceros. Para grafos dispersos, la matriz de adyacencia ocupa el mismo espacio que si el grafo tuviera muchos arcos (grafo denso). Cuando esto ocurre, se elige la representación del grafo con listas enlazadas, denominadas *listas de adyacencia*.

Las listas de adyacencia son una estructura multienlazada formada por una tabla directorio en la que cada elemento representa un vértice del grafo, del cual emerge una lista enlazada con todos sus vértices adyacentes. Es decir, cada lista representa los arcos con el vértice origen del nodo de la lista directorio, por eso se llama lista de adyacencia.

Ejemplo 15.4

La Figura 15.10 representa un grafo dirigido. La representación mediante listas de adyacencia se encuentra en la Figura 15.11.

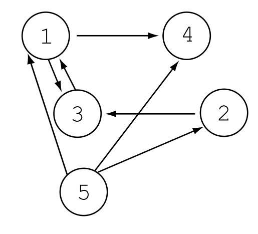


Figura 15.10 Grafo dirigido

Si se analiza el vértice `5`, es adyacente a los vértices `1, 2` y `4`; por ello su lista de adyacencia consta de tres nodos, cada uno con el vértice destino que forma el arco. El vértice `4` no es origen de ningún arco, su lista de adyacencia está vacía.

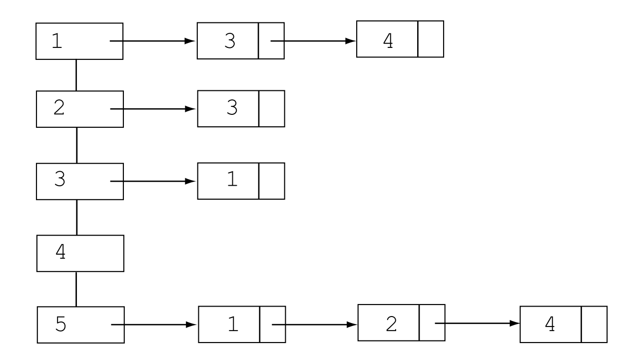


Figura 15.11 Listas de adyacencia del grafo de la Figura 15.10

15.3.1. Clase `GrafoAdcia`

Cada elemento de la tabla directorio es un vértice del grafo que guarda el identificador del vértice, su número y la lista de adyacencia. La clase `VérticeAdy` declara estos datos. Los métodos de la clase tienen la misma funcionalidad que la clase `Vertice`.

```
String nombre;
int numVertice;
Lista lad; // lista de adyacencia
           // constructor de la clase
public VerticeAdy(String x)
{
   nombre = x;
   numVertice = -1:
   lad = new Lista();
}
```

La lista de adyacencia de un vértice `u`, consta de tantos nodos como arcos tiene por origen `u`. Un nodo de la lista contiene un objeto de la clase `Arco`, en la cual se guarda el vértice destino `v` del arco que tiene su origen en `u`; además, en los grafos valorados, el *peso* asociado al arco. La clase `Arco`:

```
package Grafo;
public class Arco
{
   int destino;
   double peso;
   public Arco(int d)
   {
      destino = d;
   }

   public Arco(int d, double p)
   {
```

```
        this(d);
        peso = p;
    }
    public int getDestino()
    {
        return destino;
    }

    public boolean equals(Object n)
    {
        Arco a = (Arco)n;
        return destino == a.destino;
    }
}
```

La clase `GrafoAdcia` define la tabla de vértices con sus respectivas listas de adyacencia. Implementa las operaciones básicas para crear un grafo: añadir un vértice, insertar un arco, dar de baja un vértice y sus arcos...

```
package Grafo;
public class GrafoAdcia
{
    int numVerts;
    static int maxVerts = 20;
    VerticeAdy [] tablAdc;
```

Constructor

Crea la tabla de listas de adyacencia para un número de vértices preestablecido:

```
public GrafoAdcia(int mx)
{
    tablAdc = new VerticeAdy[mx];
    numVerts = 0;
    maxVerts = mx;
}
```

15.3.2. Realización de las operaciones con listas de adyacencia

Los métodos que implementan las operaciones forman la interfaz principal de la clase `GrafoAdcia`[1].

Nuevo vértice

Añade un nuevo vértice a la lista directorio. Si el vértice ya está en la tabla no hace nada, devuelve control; si es nuevo, se asigna a continuación del último. La implementación no considera que pueda haber *overflow*; el lector puede implementar un método auxiliar que duplique la tabla si ésta se llena. El tiempo de la operación depende de la búsqueda, en el peor caso es lineal, `O(n)` siendo `n` el número de nodos.

Arista

Esta operación da entrada a un arco del grafo. El método tiene como entrada el vértice origen y el vértice destino. El método `adyacente()` determina si la arista ya está en la lista de adyacencia,

[1] La codificación completa de la clase se encuentra en la página web del libro.

y, por último, el *Arco* se inserta en la lista de adyacencia del nodo origen. La inserción se hace como primer nodo para que el tiempo sea constante, `O(1)`.

Otra versión del método recibe, directamente, los números de vértices que forman los extremos del arco.

Para añadir una arista en un grafo valorado, se debe asignar el *factor de peso* al crear el *Arco*.

Borrar arco

La operación consiste en eliminar el nodo de la lista de adyacencia del vértice origen que contiene el arco del vértice destino. Una vez encontrada la dirección de ambos vértices en la lista de nodos se accede a la correspondiente lista de adyacencia para proceder a borrar el nodo que contiene el vértice destino.

Adyacente

La operación determina si dos vértices, `v1` y `v2`, forman un arco. En definitiva, si el vértice `v2` está en la lista de adyacencia de `v1`. El método `buscarLista()` devuelve la dirección del nodo que contiene al arco, si no está devuelve `null`.

Borrar vértice

Eliminar un vértice de un grafo es una operación que puede ser considerada costosa en tiempo de ejecución, ya que supone acceder a cada uno de los elementos de la multiestructura. En primer lugar, hay que buscar la dirección (puntero) del vértice en la lista directorio. En segundo lugar, eliminar cada uno de los nodos de la correspondiente lista de adyacencia. Y por último, recorrer cada lista de adyacencia y si encuentra algún arco con el vértice que se está borrando, se elimina de la lista. (Se deja como ejercicio la implementación de la operación.)

15.4. RECORRIDO DE UN GRAFO

En general, *recorrer una estructura* consiste en visitar (procesar) cada uno de los nodos a partir de uno dado. Se puede recorrer una lista, o un árbol en, por ejemplo, *preorden* partiendo del nodo raíz. De igual forma, *recorrer un grafo* consiste en visitar todos los vértices alcanzables a partir de uno dado. Muchos de los problemas que se plantean con los grafos exigen examinar las aristas o arcos de que consta y procesar los vértices.

Siendo `V` el conjunto de vértices del grafo, los algoritmos de *recorrido* de grafos parten de un nodo, `v`, y mantienen un conjunto de nodos *marcados* (*procesables*), `W`, que inicialmente es el nodo o vértice de partida, `v`. Cada *pasada* del algoritmo retira un nodo `w`, del conjunto `W`, que se procesa y para cada una de las aristas cuyo origen es `w`, `(w,u)`, su vértice adyacente `u` se añade a `W` si no está marcado. El algoritmo continúa hasta que el conjunto `W` se queda vacío, en ese momento todo vértice no marcado no es accesible desde el nodo de partida `v`. Si se necesita un recorrido completo, se puede continuar desde cualquier vértice no marcado.

> ***Nota***
>
> Hay dos formas de recorrer un grafo: *recorrido en profundidad* y *recorrido en anchura.* Si el conjunto de nodos *marcados* se trata como una cola, el recorrido es en anchura; si se trata como una pila, el recorrido es en profundidad.

15.4.1. Recorrido en anchura

Se utiliza una cola como estructura en la que se mantienen los vértices marcados que se van a procesar posteriormente. El proceso de los elementos en una cola (*primero en entrar primero en salir*) hace que, a partir del vértice de partida `v` se procesen primero todos los vértices adyacentes a `v`, después los adyacentes de éstos que no estuvieran ya *marcados* o *visitados*, y así sucesivamente con los adyacentes de los adyacentes.

El orden en que se visitan los nodos en el recorrido en anchura se expresa de manera más concisa en los siguientes pasos:

1. *Marcar* el vértice de partida `v`.
2. *Meter* en la cola el vértice de partida `v`.
3. Repetir los pasos 4 y 5 hasta que se cumpla la condición *cola vacía*.
4. *Quitar* el nodo frente de la cola, `w`, visitar `w`.
5. *Meter* en la cola todos los vértices adyacentes a `w` que no estén marcados y, a continuación marcar esos vértices.
6. Fin del recorrido.

Ejemplo 15.5

Recorrer en anchura el grafo dirigido de la Figura 15.12.

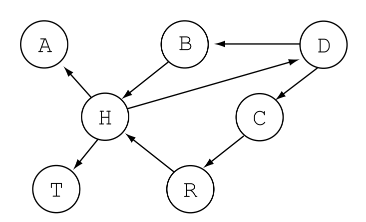


Figura 15.12 Grafo dirigido

El recorrido se inicia a partir del nodo `D`, se marca y se mete en la cola. Iterativamente, se quita el nodo frente, se procesa, se meten en la cola sus adyacentes no marcados y se marcan. Los sucesivos elementos de la cola se muestran en la Figura 15.13. En la columna de vértices procesados el vértice en negrita es el que se procesa en esa pasada, y en la cola los vértices en cursiva son los que se meten en la cola y son marcados.

COLA	Vértices procesados
D	
B C	**D**
C H	**B**
H R	**C**
R A T	**H**
A T	**R**
T	**A**
cola vacía	**T**

Figura 15.13 Recorrido en anchura del grafo de la Figura 15.12

El recorrido en anchura, a partir del nodo `D`, del grafo de la Figura 15.12 ha accedido a todos los nodos del grafo; se puede afirmar que todos sus vértices son alcanzables desde el vértice `D`.

15.4.2. Recorrido en profundidad

La búsqueda de los vértices y aristas de un grafo en profundidad persigue el mismo objetivo que el recorrido en anchura: *visitar* todos los vértices del grafo alcanzables desde un vértice dado. Difiere este recorrido con el recorrido en anchura sólo en el orden en que se procesan los vértices adyacentes. El orden en el recorrido en profundidad es el que determina la estructura pila.

El recorrido empieza por un vértice `v` del grafo, se marca como visitado y se *mete* en la pila. Después se recorre en profundidad cada vértice adyacente a `v` no visitado; hasta que no haya más vértices adyacentes no visitados. Esta estrategia de examinar los vértices se denomina en profundidad porque la dirección de *visitar* es hacia *adelante* mientras resulta posible; al contrario que la búsqueda en anchura que primero visita todos los vértices posibles en *amplitud*.

Ejemplo 15.6

Recorrer en profundidad el grafo dirigido de la Figura 15.12.

El recorrido se inicia a partir del nodo `D`, se marca y se mete en la pila. Iterativamente, se quita el nodo cabeza, se procesa, se meten en la pila sus adyacentes no marcados y se marcan. Los sucesivos elementos de la pila se muestran en la Figura 15.14. En la columna de vértices procesados en *negrita* está el que se *visita* en esa pasada, y los vértices en cursiva son los que se meten en la pila y a la vez son marcados.

FILA	Vértices procesados
D	
B C	**D**
B R	**C**
B H	**R**
B A T	**H**
B A	**T**
B	**A**
pila vacía	**B**

Figura 15.14 Recorrido en profundidad del grafo de la Figura 15.12

15.4.3. Implementación: Clase `RecorreGrafo`

La implementación de estos algoritmos se realiza con métodos `static` que reciben como argumento el grafo (con matriz o con listas de adyacencia) y el vértice de partida del recorrido.

Recorrido en anchura

Utiliza una cola en la que se guardan los vértices adyacentes al que se ha procesado. Para determinar si un vértice está o no marcado se pueden seguir varias alternativas, una de ellas utiliza el *array* `m[]`, de tantas posiciones como vértices y con una correspondencia directa entre índice

y número de vértice, para indicar si un nodo está marcado. El estado de nodo `i` *no marcado* se establece con una *clave*, si está marcado tendrá un número finito.

Al recorrer el grafo se puede obtener cierta información relativa a los vértices. En la codificación que se realiza a continuación, se guarda en `m[i]` el número mínimo de aristas para cualquier camino, desde el vértice de partida hasta el vértice `i`. El vértice de partida, `v`, se inicializa `m[v] = 0` ya que los caminos parten desde ese vértice; las otras posiciones de `m[i]` se inicializan al valor *clave* que expresa *vértice no marcado*. La implementación realizada es para un grafo representado por su matriz de adyacencia; el vértice origen viene dado por su nombre `(String)`.

```
public static
int[]recorrerAnchura(GrafoMatriz g, String org) throws Exception
{
   int w, v;
   int [] m;

   v = g.numVertice(org);
   if (v < 0) throw new Exception("Vértice origen no existe");

   ColaLista cola = new ColaLista();
   m = new int[g.numeroDeVertices()];
   // inicializa los vértices como no marcados
   for (int i = 0; i < g.numeroDeVertices(); i++)
      m[i] = CLAVE;
   m[v] = 0;     // vértice origen queda marcado
   cola.insertar(new Integer(v));
   while (! cola.colaVacia())
   {
      Integer cw;
      cw = (Integer) cola.quitar()
      w = cw.intValue();
      System.out.println("Vértice " + g.verts[w] + "visitado");
      // inserta en la cola los adyacentes de w no marcados
      for (int u = 0; u < g.numeroDeVertices(); u++)
         if ((g.matAd[w][u] == 1) && (m[u] == CLAVE))
         {
            // se marca vertice u con número de arcos hasta el
            m[u] = m[w] + 1;
            cola.insertar(new Integer(u));
         }
   }
   return m;
}
```

Para visitar todos los vértices del grafo, una vez que ha terminado el recorrido a partir de uno dado, `v`, hay que buscar si queda algún vértice sin marcar, en cuyo caso se vuelve a recorrer a partir de él; así sucesivamente hasta que todos los vértices estén marcados.

Recorrido en profundidad

La implementación utiliza una pila para establecer el orden de recorrido. También se podría realizar una implementación recursiva para evitar la utilización de la pila. Como en el recorrido

en anchura, los vértices ya visitados se marcan en el *array* `m[]`. Ahora, el grafo está representado mediante listas de adyacencia.

```
static
public int[] recorrerProf(GrafoAdcia g, String org) throws Exception
{
   int v, w;
   PilaLista pila = new PilaLista();
   int [] m;
   m = new int[g.numeroDeVertices()];
   // inicializa los vértices como no marcados
   v = g.numVertice(org);
   if (v < 0) throw new Exception("Vértice origen no existe");
   for (int i = 0; i < g.numeroDeVertices(); i++)
      m[i] = CLAVE;
   m[v] = 0;                      // vértice origen queda marcado

   pila.insertar(new Integer(v));
   while (!pila.pilaVacia())
   {
      Integer cw;
      cw = (Integer) pila.quitar();
      w = cw.intValue();
      System.out.println("Vértice" + g.tablAdc[w] + "visitado");
      // inserta en la pila los adyacentes de w no marcados
      // recorre la lista con un iterador
      ListaIterador list = new ListaIterador(g.tablAdc[w].lad);
      Arco ck;
      do
      {
         int k;
         ck = (Arco) list.siguiente();
         if (ck != null)
         {
            k = ck.getDestino();  // vértice adyacente
            if (m[k] == CLAVE)
            {
               pila.insertar(new Integer(k));
                  m[k] = 1;        // vértice queda marcado
            }
         }
      } while (ck != null);
   }
   return m;
}
```

15.5. CONEXIONES EN UN GRAFO

Al modelar un conjunto de objetos y sus relaciones mediante un grafo, una de las cuestiones que generalmente interesa conocer es si desde cualquier vértice se puede acceder al resto de los vértices del grafo, es decir, si todos los vértices están conectados o, simplemente, si el grafo es conexo. Para un grafo dirigido, la conectividad entre todos los vértices se denomina: *grafo fuertemente conexo.*

15.5.1. Componentes conexas de un grafo

Un *grafo* no dirigido G es *conexo* si existe un camino entre cualquier par de vértices que forman el grafo. En el caso de que el grafo no sea conexo resulta interesante determinar aquellos subconjuntos de vértices que mutuamente están conectados; es decir, las componentes conexas del mismo. El algoritmo para determinar las componentes conexas de un grafo G se basa en el recorrido del grafo (en *anchura* o en *profundidad).* Los pasos a seguir son los siguientes:

1. Realizar un recorrido del grafo a partir de cualquier vértice w.
2. Si en el recorrido se han *marcado* todos los vértices, entonces el grafo es conexo.
3. Si el grafo no es conexo, los vértices marcados forman una componente conexa.
4. Se toma un vértice no marcado, z, y se realiza de nuevo el recorrido del grafo a partir de z. Los nuevo vértices marcados forman otra componente conexa.
5. El algoritmo termina cuando todos los vértices han sido marcados (visitados).

15.5.2. Componentes fuertemente conexas de un grafo

Un grafo dirigido fuertemente conexo es aquel en el cual existe un camino entre cualquier par de vértices del grafo. De no ser fuertemente conexo se pueden determinar componentes fuertemente conexas del grafo. La Figura 15.15 muestra un grafo dirigido y sus dos componentes fuertemente conexas.

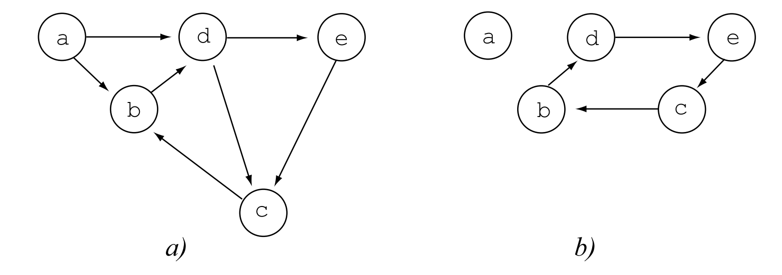


Figura 15.15 *a)* Grafo dirigido; *b)* componentes fuertes del grafo

El recorrido en profundidad a partir de un vértice dado permite diseñar un algoritmo para encontrar si un grafo es fuertemente conexo o, en su caso, determinar las componentes fuertemente conexas. A continuación se indican los pasos que sigue:

1. Obtener el conjunto de vértices descendientes del vértice de partida v, D(v), incluido el propio vértice v.
2. Obtener el conjunto de ascendientes de v, A(v), incluido el propio vértice v.
3. Los vértices comunes del conjunto de descendientes y ascendientes de v: D(v) ∩ A(v), es el conjunto de vértices que forman la componente fuertemente conexa a la que pertenece v. Si este conjunto es el conjunto de todos los vértices del grafo, entonces es fuertemente conexo.
4. Si no es un grafo fuertemente conexo se selecciona un vértice cualquiera, w, que no esté en ninguna componente fuerte de las encontradas (w ≠ D(v) ∩ A(v)) y se procede de la misma manera, es decir, se repite a partir del primer paso hasta obtener todas las componentes fuertes del grafo.

Para buscar los vértices descendientes de `v` se realiza un recorrido en profundidad del grafo a partir de `v`. Los vértices que son alcanzados se guardan en el conjunto `D`.

Los vértices ascendientes de `v` son aquellos desde los que se puede alcanzar a `v`. Por consiguiente, se obtiene el grafo inverso, cambiando la dirección de las aristas, y a continuación se recorre en profundidad el grafo inverso a partir de `v`. Los vértices alcanzados en el recorrido son los ascendientes de `v`.

Al recorrer el grafo de la Figura 15.15 en profundidad, a partir del vértice `d`, el conjunto de vértices que alcanza es: `{d, c, b, e}`. Repitiendo el recorrido en el grafo inverso, Figura 15.16, se obtiene los vértices ascendientes: `{d, b, c, e}`. Los vértices comunes `(d, b, c, e)`, forman una componente fuertemente conexa.

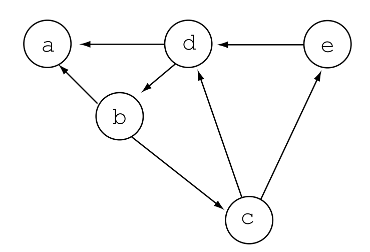


Figura 15.16 Grafo inverso del de la Figura 15.15

Ejercicio 15.2

Se tiene un grafo dirigido de n nodos, representado por su matriz de adyacencia, A. Se desea determinar si el grafo es fuertemente conexo, o bien en el caso de que no lo sea, encontrar las componentes fuertemente conexas. Las compones fuertes se mostrarán por pantalla.

El algoritmo recorre el grafo a partir de un vértice `v`, y también recorre el grafo inverso a partir del mismo vértice. El método `grafoInverso()` crea el grafo inverso, cambiando la dirección de los arcos originales.

Se parte de cualquier vértice, por ejemplo el primero, para encontrar el conjunto de vértices que están interconectados. Si son todos, el grafo es fuertemente conexo; en caso contrario, se repite el proceso a partir de un vértice que no esté formando parte de componentes anteriores. El recorrido en profundidad a partir de `v` encuentra los vértices descendientes de `v`; el recorrido se repite a partir del mismo vértice, pero con el grafo inverso, los vértices marcados forman los vértices ascendientes de `v`. Los vértices marcados en ambos recorridos forman una componente fuerte del grafo.

Al haber una correspondencia biunívoca entre el número de vértices y los índices de los *arrays* de vértices, `descendientes[]` y `ascendientes[]`, si un vértice `i` está presente se activa (se pone a *true*) la misma posición del *array;* de esa forma se guarda el vértice en el correspondiente conjunto. Esto facilita la operación de intersección (vértices comunes) ya que, simplemente, si se cumple `ascendientes[i] && descendientes[i]` el nodo `i` pertenece a ambos conjuntos. Además, en el *array* `bosque[]` se marcan los vértices que ya están formando parte de una componente conexa. El método `todosArboles()` devuelve un vértice que todavía no forma parte de componente conexa, y el proceso termina cuando devuelve el valor clave `-1`.

El método que da entrada a los componentes del grafo: vértices y arcos, se deja como ejercicio al lector.

```
import java.io.*;
import Grafo.*;
public class ComponentesFuertes
{
   static BufferedReader entrada = new BufferedReader(
                              new InputStreamReader(System.in));
   static final int CLAVE = 0xffff;
   public static void main(String [] a)throws Exception
   {
      int n, i, v;
      GrafoMatriz ga;
      GrafoMatriz gaInverso;

      System.out.print("Número de vértices del grafo: ");
      n = Integer.parseInt(entrada.readLine());

      ga = new GrafoMatriz(n);
      gaInverso = new GrafoMatriz(n);
      int []m = new int [n];
      int []descendientes = new int[n];
      int []ascendientes = new int[n];
      int []bosque = new int[n];

      entradaGrafo(ga, n);
      grafoInverso(ga, gaInverso, n);
      Vertice [] vs = new Vertice[n];
      vs = ga.vertices();
      for (i = 0; i < n; i++)
         bosque[i] = 0;

      v = 0;   // vértice de partida
      do {
         m = RecorreGrafo.recorrerProf(ga, vs[v].nomVertice());
         // se obtiene conjunto de vértices descendientes
         for (i = 0; i < n; i++)
         {
            descendientes[i] = m[i]!= CLAVE ? 1 : 0;
         }
         // recorre el grafo inverso y obtiene ascendientes
         m = RecorreGrafo.recorrerProf(gaInverso, vs[v].nomVertice());
         // se obtiene conjunto de vértices descendientes
         for (i = 0; i < n; i++)
         {
            ascendientes[i] = m[i]!= CLAVE ? 1 : 0;
         }
         System.out.print("\nComponente conexa { ");
         for (i = 0; i < n; i++)
         {
            if (descendientes[i] * ascendientes[i] == 1)
            {
               System.out.print(" " + vs[i].nomVertice());
               bosque[i] = 1;
            }
         }
         System.out.println(" }");
```

```
         // vértice a partir del cual se obtiene otra componente
         v = todosArboles(bosque,n);
      } while (v != -1);
   } // fin del método main
   static void
   grafoInverso(GrafoMatriz g, GrafoMatriz x, int n) throws Exception
   {
      Vertice [] vr = g.vertices();
      for (int i = 0; i < n; i++)
         x.nuevoVertice(vr[i].nomVertice());
      for (int i = 0; i < n; i++)
         for (int j = 0; j < n; j++)
          if (g.adyacente(i,j)) x.nuevoArco(j,i);
   }
   static int todosArboles(int [] bosque, int n)
   {
      int i, w;
      w = i = -1;
      do
      {
         if (bosque[++i] == 0)
            w = i;
      } while ((i < n - 1)  && (w == -1));
      return w;
   }
   static void entradaGrafo(GrafoMatriz gr, int n)
                                         throws Exception{...}
}
```

15.6. MATRIZ DE CAMINOS. CIERRE TRANSITIVO

Dado un grafo G, un camino de longitud n desde el vértice v_0 hasta el vértice v_n es una secuencia de n+1 vértices v_0, v_1, v_2,..., v_n tal que el vértice inicial es v_0, el vértice final v_n y son adyacentes (v_i, v_{i+1}) *para* $0 \leq i \leq n$. Encontrar caminos entre un par de vértices, de una determinada longitud es una tarea relativamente sencilla, aunque poco eficiente, si se tiene la matriz de adyacencia del grafo.

Consideremos por un momento que la matriz de adyacencia, A, es de tipo boolean, la expresión $A_{i,k}$ && $A_{k,j}$ es *verdadera si y sólo si* los valores de ambos operandos lo son. Esta hipótesis implica que hay un arco desde el vértice i al vértice k y otro desde el vértice k al j. También se puede afirmar, que si $A_{i,k}$ && $A_{k,j}$ es *verdadera* existe un camino de longitud 2 desde el vértice i al j. Ese sencillo razonamiento se puede ampliar a la siguiente expresión:

$$(A_{i1} \;\&\&\; A_{1j}) \;||\; (A_{i2} \;\&\&\; A_{2j}) \;||\; \ldots \;||(A_{in} \;\&\&\; A_{nj})$$

Si la expresión es *verdadera* implica que hay al menos un camino de longitud 2 desde el vértice i al vértice j que pase por el vértice 1, o por el 2 ... o por el vértice n. Si ahora en la expresión se cambia el operador && por el operador *producto* y || por el operador *suma*, la expresión es el elemento A_{ij} de la matriz producto A^2. La conclusión es inmediata: los elementos (A_{ij}) de la matriz A^2 son distintos de cero si existe un camino de longitud 2 desde el vértice i al vértice j, $\forall$i,j = 1..n.

El razonamiento se puede extender para encontrar caminos de longitud 3; realizando el producto matricial $A^2 \times A = A^3$ se encuentran caminos de longitud 3 entre cualquier par de vértices del grafo.

A tener en cuenta

Una manera de encontrar los caminos de longitud m entre cualquier par de vértices de un grafo es mediante el producto matricial de A^{m-1} por la matriz de adyacencia A.

A continuación se siguen los mismos razonamientos para obtener caminos del grafo de la Figura 15.17, cuya matriz de adyacencia es la indicada en la citada figura:

$$A = \begin{vmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{vmatrix}$$

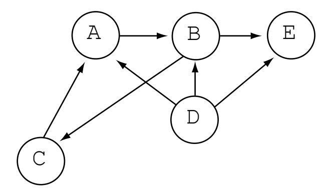


Figura 15.17 Grafo dirigido de 5 vértices

El producto matricial A x A, considerando todo valor distinto de 0 como 1 (*verdadero*):

$$A^2 = \begin{vmatrix} 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{vmatrix}$$

Si, por ejemplo, A_{15} toma el valor 1 significa que hay un camino de longitud 2, desde A - E, formado por los arcos: (A → B → E).

De igual manera, el producto matricial $A^2 \times A$:

$$A^3 = \begin{vmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{vmatrix}$$

Si nos fijamos en A_{41}, su valor es 1 ya que hay un camino longitud 3 desde D - A, formado por los arcos: (D → B → C → A).

Realizando el producto matricial, el valor almacenado en cualquier posición, A_{ij}, no sólo indica si hay camino de longitud `m`, sino además el número de caminos de dicha longitud entre los vértices `i` y `j`.

A recordar

La forma de obtener el número de caminos de longitud `k` entre cualquier par de vértices de un grafo es obtener el producto matricial A^2, A^3 ... A^k. Entonces, el elemento $A_k(i,j)$ contiene el número de caminos de longitud `k` desde el vértice `i` hasta el vértice `j`.

Ejecución

La eficiencia del algoritmo para encontrar el número de caminos de longitud `k` es muy pobre. El producto matricial se realiza con tres bucles anidados, complejidad cúbica $O(n^3)$, además, este producto se realiza `k-1` veces.

15.6.1. Matriz de caminos y cierre transitivo

Sea `G` un grafo con `n` vértices, la matriz de caminos de `G` es la matriz `P` de `n*n` elementos, definida como:

$$P_{ij} \begin{cases} 1 & \textit{si hay un camino desde } (v_i,\ v_j) \\ 0 & \textit{si no hay camino desde } (v_i,\ v_j) \end{cases}$$

Se encuentra la siguiente relación entre la matriz de caminos `P`, la matriz de adyacencia `A` y las sucesivas potencias de `A` para encontrar los caminos de longitud `m`:

"dada la matriz $B_n = A + A_2 + A_3 + \ldots + A_n$, la matriz de caminos `P` es tal que un elemento $P_{ij} = 1$ *si y sólo si* $B_n(i,j) = 1$ y en otro caso $P_{ij} = 0$ ".

El **cierre transitivo** o contorno transitivo de un grafo `G` es otro grafo, `G´`, con los mismos vértices y cuya matriz de adyacencia es la matriz de caminos del grafo `G`.

15.7. PUNTOS DE ARTICULACIÓN DE UN GRAFO

Un *punto de articulación* de un grafo no dirigido es un vértice `v` que tiene la propiedad de que si se elimina junto a sus arcos, el componente conexo en que está el vértice se divide en dos o más componentes. Por ejemplo, el grafo de la Figura 15.18 tiene dos puntos de articulación, el vértice `A` y el vértice `C`. En la Figura 15.18*b* se observa que al retirar el vértice `A` y sus arcos, el grafo se divide en dos componentes conexas: `{C,B,E,F}` y `{D}`. De igual forma, se observa en la Figura 15.18*c* que al eliminar el vértice `C` el grafo se divide en dos componentes conexas: `{B,E,F}` y `{A,D}`. Sin embargo, al suprimir cualquier otro vértice del grafo, éste no se divide en componentes.

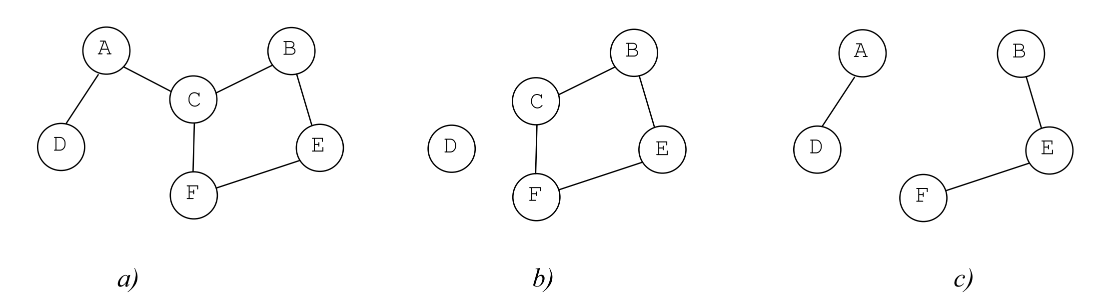


Figura 15.18 *a)* Grafo origen; *b)* después de quitar el nodo A; *c)* después de quitar el nodo C

Se estudian los *puntos de articulación* debido a que los grafos tiene propiedades relativas a ellos. Así, un grafo sin puntos de articulación se dice que es un grafo *biconexo*. De no ser el grafo biconexo, es interesante encontrar componentes biconexos. Un grafo tiene *conectividad* `k` si la eliminación de `k-1` vértices cualesquiera no divide al grafo en componentes conexas (no lo desconecta).

> ***A considerar***
>
> El estudio de los puntos de articulación de grafos, como las redes, es importante porque determinan el grado de *conectividad* del grafo y cuanto mayor es la conectividad del grafo, mayor probabilidad tiene de mantener la estructura ante el *fallo* (eliminación) de alguno de sus vértices.

15.7.1. Búsqueda de los puntos de articulación

El algoritmo de búsqueda se basa en el recorrido en profundidad para encontrar todos los puntos de articulación. Los sucesivos vértices por los que se pasa en el recorrido en profundidad de un grafo se puede representar mediante un árbol de expansión. La raíz del árbol es el vértice de partida, `A` y cada arco del grafo será una arista en el árbol.

Se aprovecha el recorrido para encontrar aristas del grafo *hacia adelante* y aristas *hacia atrás*. Así, si en el recorrido por los vértices adyacentes de `v`, arcos `(v,u)`, el vértice `u` no es visitado, entonces `(v,u)` es un arco *hacia delante* y pasa a ser una arista del árbol. Si el vértice `u` es ya visitado pero no se han terminado de procesar todos sus adyacentes, entonces `(v,u)` es un arco *hacia atrás*, no será una arista verdadera del árbol, y por ello se dibuja con una línea discontinua.

La Figura 15.19 muestra un grafo no dirigido y el árbol al que da lugar el recorrido en profundidad a partir del vértice `A`. El recorrido empieza visitando los vértices `B, C, D` que forman las correspondientes aristas del árbol, Figura 15.19*b*. Los vértices adyacentes de `D` no procesados son `E, F` y `A` del que no se ha procesado todos sus adyacentes (el vértice `D`). Los arcos `(D,E),(D,F)`forman las correspondientes aristas del árbol; el arco `(D,A)`es considerado *hacia atrás,* ya que, `A`, fue marcado anteriormente. En el árbol de la Figura 15.19*b* se señala la arista `D - A` con línea discontinua.

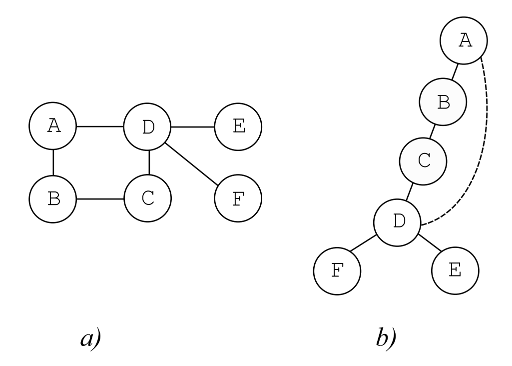


Figura 15.19 *a)* Grafo no dirigido; *b)* árbol de expansión

En el recorrido para formar el árbol se numeran los nodos del árbol, y así se obtiene el orden en que han sido visitados. El algoritmo para encontrar los puntos de articulación de un grafo conexo sigue estos pasos:

1. Recorrer el grafo en profundidad a partir de cualquier vértice. Se numeran en el orden en que son visitados los vértices y esta numeración queda asociada a cada vértice con `num(v)`.
2. Para cada nodo `v` del árbol obtenido en el recorrido se determina el vértice de numeración *más baja*, denominado `bajo(v)`, que es alcanzable desde `v` a través de cero o más aristas del árbol y como mucho una arista *hacia atrás*.
3. Una vez se tienen los valores `num(v)` y `bajo(v)`, se determinan los puntos de articulación aplicando estas reglas:
   - La raíz del árbol (vértice de partida) es punto de articulación *si y sólo* si tiene dos o más hijos.
   - Cualquier otro vértice `w` es punto de articulación *si y sólo si* `w` tiene al menos un *hijo* `u` tal que `bajo(u) ≥ num(w)`.
4. Fin del algoritmo.

En el paso 2 se introduce el concepto de numeración *más baja*, `bajo(v)`, magnitud asociada a cada vértice. Su cálculo se expresa matemáticamente como el mínimo de los siguientes tres valores:

a) `num(v)`.

b) El menor valor de `bajo()` para los vértices `a` de las aristas *hacia atrás* `(v,a)` del árbol.

c) El menor valor de `bajo()` para los vértices `w` de las aristas `(v,w)` del árbol.

En el Ejemplo 15.7 se obtienen las *numeraciones*, `num()` y `bajo()`, de los vértices del grafo no dirigido de la Figura 15.20. Aplicando las reglas descritas en el paso 3 del algoritmo, tiene dos puntos de articulación, el vértice `A` y el `C`.

Ejemplo 15.7

Dado el grafo de la Figura 15.20a encontrar las numeraciones de cada uno de sus vértices y comprobar que coinciden con las señaladas en 15.20b.

Los vértices se numeran como `0, 1, ... 5` correspondiéndose con `A .. F`. El recorrido empieza en `A` (vértice `0`), su numeración es `num(0) = bajo(0) = 1`. Los vértices que siguen en el recorrido, `D` y `C`, tienen como numeración `num(3) = bajo(3) = 2` y `num(2) = bajo(2)`

= 3. Los siguientes vértices por los que se pasa en el recorrido son, F, E y B; tienen, respectivamente, num(5) = 4, num(4) = 5, num(1) = 6. El nodo B, Figura 15.20*b* tiene una arista *hacia atrás* cuyo destino es C, por consiguiente se puede calcular su magnitud bajo() como mínimo(num(1),num(3))= 3. Entonces, bajo(1) = 3; como el vértice E tiene de descendiente a B, también bajo(4) = 3 y lo mismo con el vértice C.

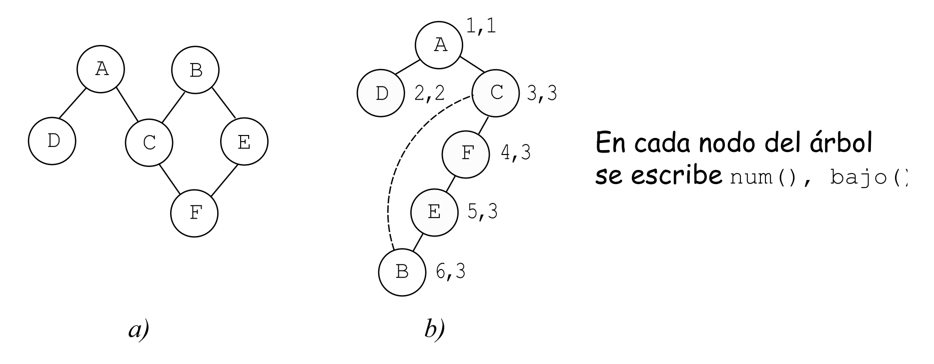


Figura 15.20 *a)* Grafo no dirigido; *b)* árbol de expansión con *numeraciones*

15.7.2. Implementación

En esta ocasión se implementa el algoritmo recursivamente. La forma de asignar los correspondientes valores de num(v) a cada vértice del grafo, consiste en incrementar, en cada llamada recursiva, a la función de recorrido el contador de llamadas, que es la magnitud num(v) para el vértice *actual*. A la vez, en el *array* arista[] se guarda v en la posición que se corresponde con el siguiente vértice adyacente, w, por el que seguirá el recorrido: arista[w] = v; los vértices (v,w) forman una arista del árbol de expansión.

Con arista[] se determinan las aristas hacia atrás aplicando este razonamiento: *si al analizar los vértices adyacentes de v resulta que "si w está ya visitado y arista[v] != w" es que (v,w) es una arista hacia atrás*.

El método recibe el grafo, no dirigido y conexo, con su matriz de adyacencia. La implementación utiliza el *array* visitado[], para marcar un vértice cuando se *pase* por él (se *visita*).

```
static void puntosArticulacion(
   GrafoMatriz g, int v, int []num, int paso,
   boolean [] visitado, int [] arista, int []bajo) throws Exception
{
   visitado[v] = true;
   num[v] = ++paso;
   bajo[v] = num[v];        // valor inicial para cálculo de bajo()
   for (int w = 0; w < g.numeroDeVertices(); w++)
   {
      if (g.adyacente(v,w))  // adyacente w
      {
         if (!visitado[w])
         {
            arista[w] = v;  // arista del árbol de expansión
            puntosArticulacion(g, w, num, paso, visitado,
                                    arista, bajo);
            if (bajo[w] >= num[v]) // v cumple la regla 3
```

```
                System.out.println("Vértice " + v +
                                         " es punto de articulación");
              bajo[v] = Math.min(bajo[v], bajo[w]);
            }
            else if (arista[v]!= w)                  // arco hacia atrás
              bajo[v] = Math.min(bajo[v],num[w]);
          }
        }
      }
```

Los *arrays* deben de crearse antes de llamar al método; además, el *array* `visitado` se inicializa a `false`:

```
int [] num = new int [g.numeroDeVertices()];
int [] bajo = new int [g.numeroDeVertices()];
int [] arista = new int [g.numeroDeVertices()];
boolean [] visitado = new boolean[g.numeroDeVertices()];

for (i = 0; i < g.numeroDeVertices(); i++)
   visitado[i] = false;
```

RESUMEN

Un grafo `G` consta de un par de conjuntos `(G = (V,E))`: conjunto `V` de vértices o nodos y conjunto `E` de *aristas* (que conectan dos vértices). Si las parejas de vértices que forman una arista no están ordenadas, `G` se denomina *grafo no dirigido;* si los pares están ordenados, entonces `G` se denomina grafo *dirigido*. El término *grafo dirigido* se suele también denominar como *dígrafo* y el término grafo sin calificación significa *grafo no dirigido.*

El método natural para dibujar un grafo es representar los vértices como puntos o círculos y las aristas como segmentos de líneas o arcos que conectan los vértices. Si el grafo está *dirigido*, entonces los segmentos de línea o arcos tienen puntas de flecha que indican la dirección.

Los grafos se implementan de dos formas: *matriz de adyacencia* y *listas de adyacencia*. Cada una tiene sus ventajas y desventajas relativas. La elección depende, entre otros factores, de la densidad del grafo; para grafos *densos*, con muchos arcos, se recomienda representarlos con la *matriz de adyacencia*. Los grafos poco densos y que experimenten modificaciones en sus componentes se representan con *listas de adyacencia*.

Dos vértices de un grafo *no dirigido* se llaman adyacentes si existe una *arista* desde el primero al segundo. Un *camino* es una secuencia de vértices distintos, cada uno adyacente al siguiente. Un *ciclo* es un *camino* que contiene al menos tres vértices, de tal modo que el último vértice en el camino es *adyacente* al primero. Un grafo se denomina *conectado* si existe un camino desde un vértice a cualquier otro vértice.

Un grafo *dirigido* se denomina *conectado fuertemente* si hay un *camino* dirigido desde un vértice a cualquier otro. Si se suprime la dirección de los arcos y el grafo *no dirigido* resultante se conecta, se denomina grafo dirigido *débilmente conectado.*

El *recorrido* de un grafo *visita* de cada uno de sus vértices, puede ser, en analogía con los árboles, *recorrido en profundidad* y *recorrido en anchura*. El *recorrido en profundidad* es una generalización del recorrido en *preorden* de un árbol. El *recorrido en anchura* visita los vértices por *niveles*, al igual que el recorrido por anchura de un árbol y permite encontrar el número mínimo de aristas para alcanzar un vértice cualquiera desde un vértice de partida.

Los *puntos de articulación* de un grafo *conexo* son aquellos vértices que si se retiran del grafo, junto a sus aristas, dividen a éste en dos *componentes conexas*. El algoritmo que permite encontrar los *puntos de articulación*, construye el *árbol de expansión* del grafo a partir de un vértice origen; recorre en profundidad el grafo buscando *aristas de árbol* y *aristas hacia atrás*.

EJERCICIOS

15.1. Dado el grafo no dirigido, `G` (Figura 15.21).

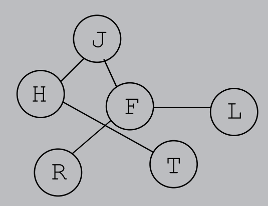


Figura 15.21 Grafo no dirigido `G`

a) Describir `G` formalmente en términos del conjunto de nodos, `V`, y del conjunto `A` de arcos.
b) Encontrar el grado de cada nodo.

15.2. Dado el grafo dirigido, `G` (Figura 15.22).

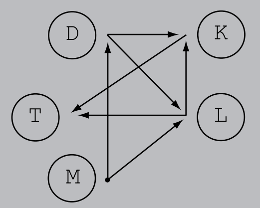


Figura 15.22 Grafo dirigido `G`

a) Describir `G` formalmente en términos de su conjunto de nodos, `V`, y de su conjunto `A` de arcos.
b) Encontrar el grado de entrada y el grado de salida de cada vértice.
c) Escribir la secuencia de vértices que forman los caminos simples del vértice `M` al vértice `T`.

15.3. Dado el grafo no dirigido, `G` (Figura 15.23).

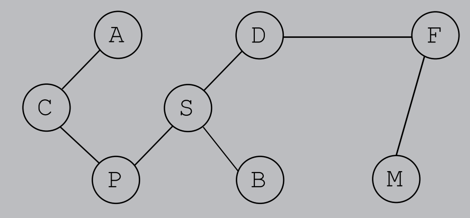


Figura 15.23 Grafo no dirigido `G`

a) Escribir la secuencia de vértices que forman los caminos simples del vértice `A` al vértice `F`.
b) Encontrar el camino más corto (en cuanto a número de aristas) del vértice `C` al vértice `D`.
c) ¿Es un grafo conexo?

15.4. Dado el grafo dirigido y valorado, `G` (Figura 15.24).

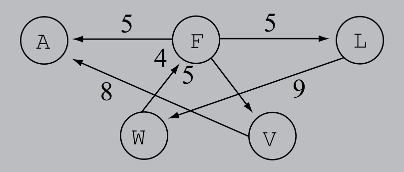


Figura 15.24 Grafo valorado

a) Escribir la matriz de pesos del grafo.
b) Representar el grafo mediante listas de adyacencia.

15.5. Un grafo está formado por los vértices `V = {A, B, C, D, E}`, su matriz de adyacencia, suponiendo los vértices numerados del 0 al 4 respectivamente:

$$M = \begin{vmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 \end{vmatrix}$$

a) Dibujar el grafo que le corresponde.
b) Representar el grafo mediante listas de adyacencia.

15.6. Un grafo no dirigido conexo tiene la propiedad de ser biconexo si no hay ningún vértice que al suprimirlo del grafo haga que este se convierta en no conexo.

a) Dibujar un grafo de 6 nodos biconexo.
b) Determinar si los grafos de la Figura 15.21 y 15.22 son biconexos.

15.7. Dado el grafo no dirigido del ejercicio 15.5 realizar el recorrido en profundidad a partir del vértice `C`.

15.8. Dado el grafo no dirigido del ejercicio 15.5 realizar el recorrido en anchura a partir del vértice `C` y la longitud de los caminos mínimos a los demás vértices.

15.9. En la Figura 15.25 se representan dos grafos dirigidos. Teniendo en cuenta que un grafo dirigido se considera *acíclico* si no tiene ciclos, también se denominan **gda**, indicar si los grafos de la figura son **gda**, en el caso de no ser **gda**, buscar los ciclos.

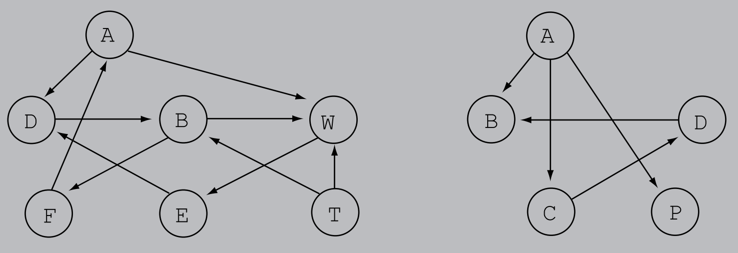


Figura 15.25 Grafos dirigidos

15.10. Dado el grafo de la Figura 15.26, encontrar la componentes fuertemente conexas.

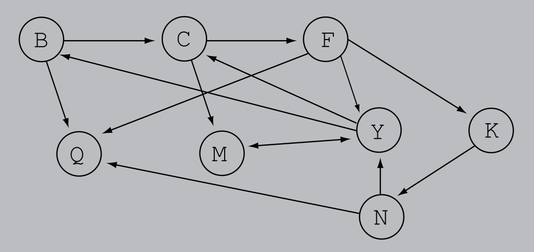


Figura 15.26 Grafo dirigido

15.11. Dado el grafo `G` de la Figura 15.27:

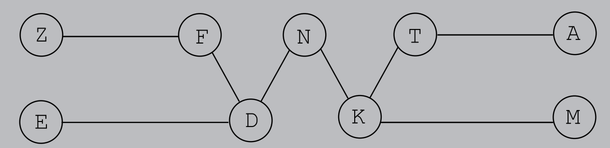


Figura 15.27 Grafo no dirigido

a) Escribir la matriz de adyacencia del grafo.
b) Escribir la matriz de caminos de `G`.

15.12. Dado un grafo dirigido en el que los vértices son números enteros positivos y el par `(x,y)` es una arista en el caso de que `x-y` sea múltiplo de `3`.

a) Representar el grafo formado por los vértices 3 al 14.
b) Determinar el grado de entrada y el grado de salida de cada vértice.

15.13. En los grafos no dirigidos de las figuras 15.23 y 15.27:

a) Dibujar los correspondientes árboles de expansión. Asociar a cada nodo su *numeración* `num()` y `bajo()`.
b) Encontrar los puntos de articulación.

15.14. Dibujar un grafo dirigido en el que los vértices son números enteros positivos del 3 al 15 para cada una de las siguientes relaciones:

a) `v` es adyacente de `w` si `v+2w` es divisible entre 3.
b) `v` es adyacente de `w` si `10v+w < v*w`.

PROBLEMAS

15.1. Diseñar un grafo valorado con los siguientes requisitos:

- Consta de 10 vértices que son números aleatorios del `11` al `99`.
- Dos vértices `v, w,` están relacionados si `v+w` es múltiplo de 3.

a) Escribir un programa en el que se genere un grafo con las condiciones descritas. El grafo se ha de representar con una matriz de adyacencia.
b) Añadir al programa los métodos necesarias para determinar la matriz de caminos utilizando las potencias de la matriz de adyacencia.

15.2. Un grafo valorado está formado por los vértices `4, 7, 14, 19, 21, 25`. Las aristas siempre van de un vértice de mayor valor numérico a otro de menor valor, y el peso es el módulo del vértice origen y el vértice destino.

a) Escribir un programa que represente el grafo con listas de adyacencia.
b) Añadir al programa el código necesario para realizar un recorrido en anchura desde un vértice dado.

15.3. Una región está formada por 12 comunidades. Se establece la relación de desplazamiento de personas en las primeras horas del día. Así, la comunidad `A` está relacionada con la comunidad `B` si desde `A` se desplazan `n` personas a `B`, de igual forma puede haber relación entre `B` y `A` si se desplazan `m` personas de `B` hasta `A`.

a) Escribir un programa que represente el grafo descrito mediante listas de adyacencia.
b) Determinar si el grafo formado tiene fuentes o sumideros.

15.4. Dado el grafo descrito en el Problema 15.3. Escriba un programa para representarlo mediante listas enlazadas de tal forma que cada nodo de la lista directorio contenga dos listas: una que contiene los arcos que salen del nodo, y otra que contiene los arcos que terminan en el nodo.

15.5. Un grafo en el que los vértices son regiones y los arcos relacionan dos regiones entre las cuales hay un flujo de emigrantes (tiene factor de peso), está representado mediante una lista directorio que contiene a cada uno de los vértices y de las que sale una lista circular con los vértices adyacentes. Ahora se quiere representar el grafo mediante una matriz de pesos, de tal forma que si entre dos vértices no hay arco su posición en la matriz tiene `0`, y si entre dos vértices hay arco su posición contiene el factor de peso que le corresponde.
Escribir las funciones necesarias de modo que partiendo de la representación mediante listas se obtenga la representación mediante la matriz de pesos.

15.6. Representar la información sobre un Centro de enseñanza que se enumera a continuación:

- Nombre del centro, Ubicación, Nombre del Director.
- Alumnos divididos en clases de:
  - Enseñanza Primaria: *n* grupos de un máximo de 25 alumnos.
  - Enseñanza Secundaria: *m* grupos de un máximo de 30 alumnos.
  - Bachillerato: *b* grupos con un máximo de 40 alumnos.
- Profesores de:
  - Enseñanza Primaria se asigna un profesor a cada clase.
  - Enseñanza Secundaria y Bachillerato, hay un profesor por cada asignatura del curso.
- Asignaturas: en cada curso hay un máximo de *max* asignaturas.

Se pide:

1. Lectura de los alumnos de cada uno de los grupos.
2. Lectura del profesorado de cada uno de los grupos.
3. Lectura de las notas de una clase.
4. Clasificación del alumnado de dicha clase en alumnos aprobados, suspensos y de muy bajo rendimiento (3 asignaturas suspensas o más).
5. Ordenación alfabética de una clase.
6. Ordenación alfabética del alumnado del colegio.

Nota: Evitar la presencia de información redundante. Hacer uso estructuras de datos dinámicas.

15.7. Durante el recorrido en profundidad de un grafo dirigido, cuando se recorren ciertos arcos, se llega a vértices aún sin visitar. Los arcos que llevan a vértices nuevos se conocen como arcos de árbol y forman un bosque abarcador en profundidad para el grafo dirigido dado. Además de los arcos del árbol, existen otros tipos de arcos diferentes que se llaman arcos de retroceso, arcos de avance y arcos cruzados:

- Un arco se dice que es de retroceso si va de un nodo del árbol a otro que es su predecesor.
- Un arco se dice que es de avance si va de un nodo del árbol a otro nodo del árbol ya construido, pero que es un descendiente de él.
- Un arco se dice que es cruzado si va de un nodo del árbol a otro que no está relacionado por la relación jerárquica definida en el árbol.

Escriba un programa que lea un grafo, calcule el bosque abarcador y los arcos de avance, retroceso y cruzado

15.8. Se dispone de un grafo no dirigido poco denso. Se elige la representación en memoria mediante listas de adyacencia. Escribir un programa en el que se de entrada a los vértices del grafo y sus arcos y se determine si el grafo tiene ciclos. En caso positivo, listar los vértices que forman un ciclo.

15.9. Supóngase un grafo no dirigido y conexo, escribir un programa que encuentre un camino que vaya a través de todas las aristas exactamente una vez en cada dirección.

15.10. Se denominan caminos hamiltonianos a aquéllos que contienen exactamente una vez a todos y cada uno de los vértices del grafo. Se trata de escribir un programa que lea un grafo e imprima todos sus caminos hamiltonianos.

15.11. Escriba un programa que compruebe si un grafo dirigido, leído del teclado, tiene circuitos (ciclos) mediante el siguiente algoritmo:

1. Obtener los sucesores de todos los vértices.
2. Buscar un vértice sin sucesores y *tachar* (eliminar) ese vértice donde aparezca (conjuntos de sucesores).
3. Se continúa el proceso en el paso 2 siempre que haya algún vértice sin sucesores.
4. Si todos los vértices del grafo han sido eliminados, no tienen circuitos.

CAPITULO 16

Grafos, algoritmos fundamentales

Objetivos

Con el estudio de este capítulo, usted podrá:

- Calcular el camino mínimo desde un vértice al resto de los vértices.
- Determinar si hay camino entre cualquier par de vértices de un grafo.
- Implementar los algoritmos mas importantes sobre grafos.
- Representar con un grafo ciertas relaciones entre objetos y aplicar los algoritmos que determinan su conectividad.
- Saber cuándo se aplica un algoritmo de expansión de coste mínimo y cuándo un camino de coste mínimo.

Contenido

Conceptos clave

- ♦ Árbol.
- ♦ Camino.
- ♦ Camino mínimo.
- ♦ Grafo valorado.
- ♦ Matriz de caminos.
- ♦ Ordenación topológica.
- ♦ Red.

INTRODUCCIÓN

Existen numerosos problemas que se pueden modelar en términos de grafos. Ejemplos concretos son: la planificación de las tareas que completan un proyecto, encontrar las rutas de menor longitud entre dos puntos geográficos, calcular el camino más rápido en un transporte o determinar el flujo máximo que puede llegar desde una *fuente* a una *urbanización.*

La resolución de estos problemas requiere examinar todos los nodos y las aristas del grafo que lo representan. Los algoritmos imponen implícitamente un orden en las visitas: el nodo más próximo o las aristas mas cortas, y así sucesivamente; no todos los algoritmos requieren un orden concreto en el recorrido del grafo.

Se estudian en este capítulo el concepto de ordenación topológica, los problemas del camino más corto y el concepto de árbol de expansión de coste mínimo: estos algoritmos han sido desarrollado por grandes investigadores y en su honor se conocen por sus nombre es: Dijkstra, Warshall, Prim, Kruscal, Ford-Fulkerson ...

16.1. ORDENACIÓN TOPOLÓGICA

Una de las aplicaciones de los grafos es modelar las relaciones que existen entre las diferentes tareas, *hitos,* que deben finalizar para dar por concluido un proyecto. Entre las tareas existen relaciones de precedencia: una tarea `r` precede a la tarea `t` si es necesario que se complete `r` para poder empezar `t`. Estas relaciones de precedencia se representan mediante un grafo dirigido en el que los vértices son las tareas o *hitos* y existe una arista del vértice `r` al `t` si el inicio de la tarea `t` depende de la terminación de `r`. Una vez se dispone del grafo interesa obtener una planificación de las tareas que constituyen el proyecto; en definitiva, encontrar la *ordenación topológica* de los vértices que forman el grafo.

El grafo que representa estas relaciones de precedencia es un grafo dirigido *acíclico,* de tal forma que si existe un camino de `u` a `v`, entonces, en la ordenación topológica `v` es posterior a `u`. El grafo no puede tener ciclos cuando representa relaciones de precedencia; en el caso de existir, significa que si `r` y `t` son vértices del ciclo, `r` depende de `t` y a su vez `t` depende de la terminación de `r`.

> ***A tener en cuenta***
>
> La ordenación topológica se aplica sobre grafos dirigidos sin ciclos. Es una ordenación lineal, tal que si `v` es anterior a `w` entonces existe un camino de `v` a `w`. La ordenación topológica no se puede realizar en grafos con ciclos.

La Figura 16.1 muestra un grafo acíclico que modela la estructura de *prerrequisito* de `8` cursos. Un arista cualquiera `(r,s)` significa que el curso `r` debe completarse antes de empezar el curso `s`. Por ejemplo, el curso `M21` se puede empezar sólo cuando se terminen los cursos `E11` y `T12`; en ese sentido, `E11` y `T12` son *prerrequisito* de `M21`.

Una ordenación topológica de estos cursos es cualquier secuencia de cursos que cumple los requerimientos *(prerrequisito).* Entonces, para un grafo dirigido acíclico no tiene por qué existir una única ordenación topológica.

Del grafo de requisitos de la Figura 16.1 se obtienen estas ordenaciones topológicas:

```
E11 - T12 - M21 - C22 - R23 - S31 - S32 - T41
T12 - E11 - R23 - C22 - M21 - S31 - S32 - T41
```

Un grafo `G` dirigido y sin ciclos se denomina un *gda* o grafo acíclico. Los *gda* son útiles para la representación de *ordenaciones parciales*. Una *ordenación parcial* `R` en un conjunto `C` es una relación binaria de precedencia tal que:

1. Para todo `u ∈ C`, `u` no está relacionado con sí mismo, `u R u` es falso, por tanto, la relación `R` es *irreflexiva*.
2. Para todo `u, v, w ∈ C`, si `u R v` y `v R w`, entonces `u R w`. La relación `R` es *transitiva.*

Tal relación `R` sobre `C` se llama *ordenación parcial* de `C`. La relación de inclusión en conjuntos es un ejemplo de ordenación parcial. Un grafo `G` sin ciclos se puede considerar un conjunto parcialmente ordenado.

> ***Nota***
>
> Se puede comprobar que un grafo es *acíclico* realizando un recorrido en profundidad, de tal forma que si se encuentra un arco de *retroceso* en el árbol de búsqueda, el grafo tiene al menos un ciclo.

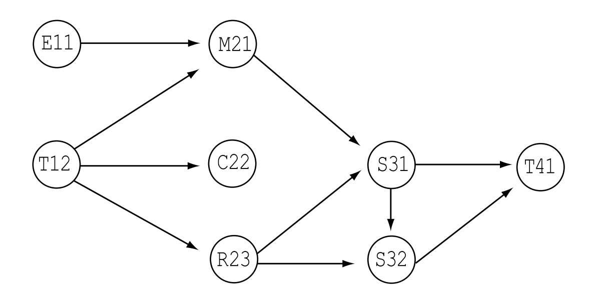


Figura 16.1 Grafo dirigido de prerrequisito

16.1.1. Algoritmo de una ordenación topológica

El algoritmo, en primer lugar, busca un vértice (una tarea) sin *predecesores* o *prerrequisitos*; es decir, no tiene arcos de entrada. Este vértice, `v`, pasa a formar parte de la ordenación `T`; a continuación, todos los arcos que salen de `v` son eliminados, debido a que el prerrequisito `v` ya se ha satisfecho. La estrategia se repite: se toma otro vértice `w` sin arcos incidentes, se incorpora a la ordenación `T` y se eliminan todos los arcos que salen de él, así sucesivamente hasta completar la ordenación.

Si un vértice `v` no tiene arcos incidentes se expresa con el grado de entrada, `gradoEntrada v)=0`. Eliminar los arcos que salen de `v` implica que `gradoEntrada(w)` de todos los vértices `w` adyacentes de `v` disminuye en `1`.

Por consiguiente, el algoritmo comienza calculando el grado de entrada de cada vértice del grafo. Los vértices cuyo grado de entrada es `0` se guardan en una *Cola*. El elemento frente de

la cola, `v`, pasa a formar parte de la ordenación `T`; a continuación, se disminuye en `1` el grado de entrada de los adyacentes de `v`, y aquellos vértices cuyo grado de entrada ha pasado a ser `0` se meten en la cola, y así sucesivamente hasta que la cola esté vacía.

Ejecución

Si al aplicar el algoritmo de ordenación topológica existen vértices del grafo que aún no han pasado a formar parte de la ordenación y la cola está vacía, entonces el grafo tiene ciclos.

16.1.2. Implementación del algoritmo de ordenación topológica

La codificación del algoritmo depende de la representación del grafo, con matriz de adyacencia o listas de adyacencia. Si el grafo tiene relativamente pocos arcos, (poco denso), la matriz de adyacencia tiene muchos ceros (es una matriz *esparcida*), y entonces el grafo se representa con listas de adyacencia. En el caso de grafos dirigidos *densos* se prefiere, por eficiencia, la matriz de adyacencia. Con independencia de la representación, se utiliza una *Cola* para almacenar los vértices con grado de entrada `0`.

La siguiente codificación representa un grafo representado con listas de adyacencia (clase *GrafoAdcia*); los vértices de la ordenación topológica se escriben por pantalla, y se guardan en el vector `T[]`.

```
import TipoCola.*;
import ListaGenerica.*;

// Método que devuelve el grado de entrada del vértice v .
//  Se suponen los vértices numerados de 0 a n-1

public static int gradoEntrada(GrafoAdcia g, int v) throws Exception
{
   int cuenta = 0;
   for (int origen = 0; origen < g.numeroDeVertices(); origen++)
   {
      if (g.adyacente(origen, v))         // arco incidente a v
           cuenta++;
   }
   return cuenta;
}

// Método para obtener una ordenación topológica.
// Muestra los vértices que pasan a formar parte de la
// ordenación, y se guardan en T[]

public static
void ordenTopologica(GrafoAdcia g, int[]T) throws Exception
{
   int []arcosInciden;
```

```
   int v, w, nvert;
   ColaLista cola = new ColaLista();
   nvert = g.numeroDeVertices();
   arcosInciden = new int[nvert];
   // grado de entrada de cada vértice
   for (v = 0; v < nvert; v++)
      arcosInciden[v] = gradoEntrada(g, v);

   System.out.println("\n Ordenación topológica ");
   for (v = 0; v < nvert; v++)
      if (arcosInciden[v] == 0)
          cola.insertar(new Integer(v));

   VerticeAdy [] vs = new VerticeAdy[nvert];
   vs = g.vertices();
   while (!cola.colaVacia())
   {
      Integer a;
      Arco elemento;
      ListaIterador itl;
      int j = 0;
      a = (Integer)cola.quitar();
      w = a.intValue();
      System.out.print(" " + vs[w].toString());
      T[j++] = w;

      itl = new ListaIterador(g.listaAdyc(w)); // iterador de lista
      // decrementa grado entrada de vértices adyacentes
      while ((elemento = (Arco)itl.siguiente())!= null)
      {
         v = elemento.getDestino();
         arcosInciden[v]--;
         if (arcosInciden[v] == 0)
            cola.insertar(new Integer(v));
      }
   }
}
```

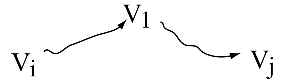
Nota

La complejidad del algoritmo *ordenación topológica* de un grafo, representado con listas de adyacencia, es `O(a+n)`, siendo `a` el número de arcos y `n` el de vértices. Si la representación del grafo es con una matriz de adyacencia, la complejidad es $O(n^2)$.

16.2. MATRIZ DE CAMINOS: ALGORITMO DE WARSHALL

Este algoritmo calcula la matriz de caminos `P` (también llamado cierre transitivo) de un grafo `G` de `n` vértices, representado por su matriz de adyacencia `A`. La estrategia que sigue el algoritmo consiste en definir, a nivel lógico, una secuencia de matrices *n-cuadradas* P_0, P_1, P_2, P_3 ... P_n; los elementos de cada una de las matrices P_k`[i,j]` tienen el valor `0` si no hay camino y `1` si existe un camino del vértice `i` y al `j`. La matriz P_0 es la matriz de adyacencia.

Los elementos P_{ij} de P_1 son igual a 1 si lo son los de la matriz P_0, o bien si se puede formar un camino desde el vértice v_i a v_j, con la ayuda del vértice 1. En definitiva, $P_1[i,j] = 1$ si lo es $P_0[i,j]$, o bien, hay un camino:



La matriz P_2 se forma a partir de P_1, añadiendo el vértice 2 para poder formar camino entre dos vértices todavía no conectados.

La diferencia entre dos matrices consecutivas P_k y P_{k-1} viene dada por la incorporación del vértice de orden k, para estudiar si es posible formar camino del vértice i y al j con la ayuda del vértice k. La descripción de cada matriz es la siguiente:

$P_0[i,j] = \begin{cases} 1 & \text{si existe un arco del vértice } i \text{ al } j. \\ 0 & \text{en otro caso.} \end{cases}$

$P_1[i,j] = \begin{cases} 1 & \text{si existe un camino simple de } v_i \text{ a } v_j \text{ que no pasa por ningún otro vértice, a no ser por el vértice } 1. \\ 0 & \text{en otro caso.} \end{cases}$

$P_2[i,j] = \begin{cases} 1 & \text{si existe un camino simple de } v_i \text{ a } v_j \text{ que no pasa por otro vértice, a no ser por los que están comprendidos entre los vértices } 1 \text{ y } 2. \\ 0 & \text{en otro caso.} \end{cases}$

$P_3[i,j] = \begin{cases} 1 & \text{si existe un camino simple de } v_i \text{ a } v_j \text{ que no pasa por otro vértice, a no ser por los que están comprendidos entre los vértices } 1 \text{ y } 3. \\ 0 & \text{en otro caso.} \end{cases}$

En cada paso se incorpora un nuevo vértice, el que coincide con el índice de la matriz P, a los anteriores para poder formar camino.

$P_k[i,j] = \begin{cases} 1 & \text{si existe un camino simple de } v_i \text{ a } v_j \text{ que no pasa por otro vértice, a no ser por los que están comprendidos entre los vértices } 1 \text{ y } k. \\ 0 & \text{en otro caso.} \end{cases}$

$P_n[i,j] = \begin{cases} 1 & \text{si existe un camino simple de } v_i \text{ a } v_j \text{ que no pasa por otro vértice, a no ser por los que están comprendidos entre los vértices } 1 \text{ y } n. \\ 0 & \text{en otro caso.} \end{cases}$

Según estas definiciones, P_0 es la matriz de adyacencia del grafo de n vértices, la matriz P_n es la matriz de caminos.

El algoritmo de Warshall encuentra una relación recurrente entre los elementos de la matriz P_k y los elementos de la matriz P_{k-1}. Un elemento $P_k[i,j] = 1$ si ocurren uno de estos dos casos:

1. Existe un camino simple de v_i a v_j del cual pueden formar parte los vértices de índice 1 al k-1 (v_1 a v_{k-1}); por consiguiente, el elemento $P_{k-1}[i,j] = 1$.

2. Existe un camino simple de v_i a v_k y otro camino simple de v_k a v_j de los cuales pueden formar parte los vértices de índice `1` al `k-1`; por consiguiente, esto equivale:

$(P_{k-1}\ [i,k] = 1)$ y $(P_{k-1}\ [k,j] = 1)$

$v_i \rightarrow \ldots \rightarrow v_j$ (1) $\qquad v_i \rightarrow \ldots \rightarrow v_k \rightarrow \ldots \rightarrow v_j$ (2)

Camino de v_i a v_j $\qquad$ *Camino* $v_i - v_k - v_j$

Camino de v_i a v_j $\qquad$ Camino de $v_i - v_k - v_j$

Figura 16.2 Posibilidad de camino de v_i a v_j

En definitiva, Warshall encuentra una relación recurrente entre la matriz P_{k-1} y P_k que permite, a partir de, la matriz de adyacencia P_0, encontrar P_n, matriz de caminos. La relación para encontrar los elementos de P_k puede expresarse como una operación lógica:

$$P_k\ [i,j] = P_{k-1}[i,j] \vee (P_{k-1}\ [i,k] \wedge P_{k-1}\ [k,j])$$

16.2.1. Implementación del algoritmo de Warshall

Se codifica el algoritmo para un grafo `G` representado por su matriz de adyacencia. El método devuelve la matriz de caminos `P`.

```
public static int [][] matrizCaminos(GrafoMatriz g) throws Exception
{
   int n = g.numeroDeVertices();
   int [][] P = new int[n][n]; // matriz de caminos
   // Se obtiene la matriz inicial: matriz de adyacencia
   for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++)
         P[i][j] = g.adyacente(i,j) ? 1 : 0;

   // se obtienen, virtualmente, a partir de P0, las sucesivas
   // matrices P1, P2, P3 ,..., Pn-1, Pn que es la matriz de caminos

   for (int k = 0; k < n; k++)
      for (int i = 0; i < n; i++)
         for (int j = 0; j < n; j++)
            P[i][j] = Math.min(P[i][j] + P[i][k] * P[k][j], 1);

   return P;
}
```

Nota sobre eficiencia

La complejidad del algoritmo de *Warshall* es cúbica, $O(n^3)$ siendo n el número de vértices. Esta característica hace que el tiempo de ejecución crezca rápidamente para grafos con, relativamente, muchos nodos.

16.3. CAMINOS MÁS CORTOS CON UN SOLO ORIGEN: ALGORITMO DE DIJKSTRA

Uno de los problemas que se plantean con relativa frecuencia es determinar la longitud del camino más corto entre un par de vértices de un grafo. Por ejemplo, determinar la mejor ruta (menor tiempo) para ir desde un lugar a un conjunto de centros de la ciudad. Para resolver este tipo de problemas se considera un grafo dirigido y *valorado*; es decir, cada arco (v_i,v_j) del grafo tiene asociado un coste c_{ij}. La longitud del camino es la suma de los costes de los arcos que forman el camino. Matemáticamente, si $v_1,v_2 \dots v_k$ es la secuencia de vértices que forman un camino:

$$Longitud\ de\ camino = \sum_{i=1}^{k-1} C_{i,j+1}$$

se pretende encontrar, entre todos los caminos, el camino de longitud o coste mínimo.

Otro problema relativo a los caminos entre dos vértices v_1, v_k, es encontrar aquel de menor número de arcos para ir de v_1 a v_k (el camino de longitud mínima en un grafo no valorado). Este problema se resuelve con una *búsqueda en anchura* a partir del vértice de partida (véase el capítulo anterior).

El algoritmo que se describe a continuación encuentra el camino más corto desde un vértice origen al resto de vértices, en un grafo con factor de peso positivo. Recibe el nombre de algoritmo de Dijkstra en honor del científico que lo expuso.

16.3.1. Algoritmo de Dijkstra

Dado un grafo dirigido G=(V,A) valorado y con factores de peso no negativos, el algoritmo de Dijkstra determina el camino más corto desde un vértice al resto de los vértices del grafo. Éste, es un ejemplo típico de algoritmo *ávido (voraz)* que selecciona en cada paso la solución más optima para resolver el problema.

Recordemos que se trabaja con un grafo valorado donde cada arco (v_i,v_j) tiene asociado un coste c_{ij}, de tal forma que si v_0 es el vértice origen, y $v_0, v_1 \dots v_k$ es la secuencia de vértices que forman el camino de v_0 a v_k, entonces $\sum_{i=0}^{k-1} C_{i,i+1}$ es la longitud del camino.

El algoritmo *voraz* de Dijkstra considera dos subconjuntos de vértices, F y V-F, donde V es el conjunto de todos los vértices. Se define la función distancia(v): coste del camino más corto del origen s a v que pasa solamente por los vértices de F y tal que el primer vértice que se añade a F es el origen s. En cada paso se selecciona un vértice v de V-F cuya *distancia* a F es la menor; el vértice v se *marca* para indicar que ya se conoce el camino mas corto de s a v. Se

siguen *marcando* nuevos vértices de `V-F` hasta que lo estén todos; en ese momento el algoritmo termina y ya se conoce el camino más corto del vértice origen `s` al resto de los vértices.

Para realizar esta estrategia *voraz,* el *array* `D` almacena el camino más corto (coste mínimo) desde el origen `s` a cada vértice del grafo. Se parte con el vértice origen `s` en `F` y los elementos D_i, del *array* `D`, con el coste o peso de los arcos (s, v_i); si no hay arco de `s` a `i` se pone *coste* ∞ . En cada paso se agrega algún vértice `v` de los que todavía no están en `F`, es decir, de `V-F`, que tenga el menor valor `D(v)`. Además, se actualizan los valores `D(w)` para aquellos vértices `w` que todavía no están en `F`, $D(w) = D(v)+c_{v,w}$ cuando este nuevo valor de `D(w)` sea menor que el anterior. De esta forma se asegura que la incorporación de `v` implica que hay un camino de `s` a `v` cuyo coste mínimo es `D(v)`.

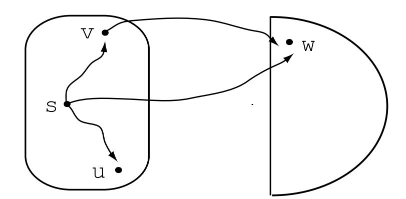


Figura 16.3 Al incorporar `v`, la distancia de `s` a `w`, `D[w]` puede mejorar

Para reconstruir el camino de *coste mínimo* que lleva de `s` a cada vértice `v` del grafo, se almacena, para cada uno de los vértices, el último vértice que hizo el coste mínimo. Entonces, asociado a cada vértice, resultan de interés dos datos: el *coste mínimo* y el último vértice del camino con el que se minimizó el *coste*.

Ejemplo 16.1

Dado el grafo dirigido y con pesos no negativos de la Figura 16.4 se desea calcular el coste mínimo de los caminos desde el vértice 1 a los otros vértices del grafo, aplicando el algoritmo de Dijkstra.

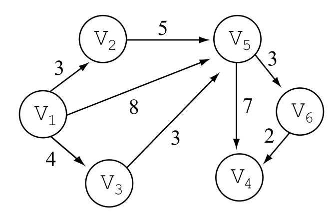


Figura 16.4 Grafo dirigido con factor de peso positivo

La matriz de pesos de los arcos, considerando peso ∞ cuando no existe arco es:

$$C = \begin{vmatrix} \infty & 3 & 4 & \infty & 8 & \infty \\ \infty & \infty & \infty & \infty & 5 & \infty \\ \infty & \infty & \infty & \infty & 3 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & 7 & \infty & 3 \\ \infty & \infty & \infty & 2 & \infty & \infty \end{vmatrix}$$

Los sucesivos valores del conjunto F (vértices *marcados*), vértice v incorporado y el vector distancia, D, que se obtienen en cada paso se representan en la siguiente tabla:

Paso	F	v	D[2]	D[3]	D[4]	D[5]	D[6]
Inicial	1		3	4	∞	8	∞
1	1,2	2	3	4	∞	8	∞
2	1,2,3	3	3	4	∞	7	∞
3	1,2,3,5	5	3	4	14	7	10
4	1,2,3,5,6	6	3	4	12	7	10
5	1,2,3,5,6,4	4	3	4	12	7	10

Por ejemplo: el camino mínimo de v_1 a v_6 es 10; la secuencia de vértices que hacen el camino mínimo: v_1 - v_3 - v_5 - v_6 .

16.3.2. Codificación del algoritmo de Dijkstra

La clase GrafMatPeso representa un grafo valorado. La matriz de pesos contiene el *coste* de cada arco, o bien INFINITO si no hay arco entre dos vértices. Además, el método pesoArco() devuelve el peso o coste de un arco (se puede consultar la clase en la página web del libro).

Se escribe la clase CaminoMinimo para implementar el algoritmo y operaciones auxiliares. Los *arrays* ultimo[] y D[] representan la información asociada a cada vértice: el último vértice en el camino y el coste mínimo, siendo el vértice el índice del *array*. El conjunto F de vértices se representa mediante un *array* de tipo boolean, de tal forma que si el vértice i se incluye en F, entonces F[i] se pone a *true* (verdadero).

```
package Grafo;

public class CaminoMinimo
{
   private int [][] Pesos;
   private int [] ultimo;
   private int [] D;
   private boolean [] F;
   private int s, n;          // vértice origen y número de vértices
   public CaminoMinimo(GrafMatPeso gp, int origen)
   {
      n = gp.numeroDeVertices();
      s = origen;
      Pesos = gp.matPeso;
      ultimo = new int [n];
      D = new int [n];
      F = new boolean [n];
   }
   public void caminoMinimos()
   {
      // valores iniciales
      for (int i = 0; i < n; i++)
```

```
      {
         F[i] = false;
         D[i] = Pesos[s][i];
         ultimo[i] = s;
      }
      F[s] = true; D[s] = 0;
         // Pasos para marcar los n-1 vértices
      for (int i = 1; i < n; i++)
      {
         int v = minimo();  /* selecciona vértice no marcado
                                                 de menor distancia */
         F[v] = true;
         // actualiza distancia de vértices no marcados
         for (int w = 1; w < n; w++)
            if (!F[w])
               if ((D[v] + Pesos[v][w]) < D[w])
               {
                  D[w] = D[v] + Pesos[v][w];
                  ultimo[w] = v;
               }
      }
   }
   private int minimo()
   {
      int mx = GrafMatPeso.INFINITO;
      int v = 1;
      for (int j = 1; j < n; j++)
         if (!F[j] && (D[j]<= mx))
         {
            mx = D[j];
            v = j;
         }
      return  v;
   }

}
```

El tiempo de ejecución del algoritmo es cuadrático, se deduce que tiene una complejidad cuadrática, $O(n^2)$, debido a los dos bucles anidados de orden n (número de vértices). Ahora bien, en el caso de que el número de arcos, a, fuera mucho menor que n^2, el tiempo de ejecución mejora representando el grafo con listas de adyacencia. Entonces puede obtenerse un tiempo O(alog n).

Recuperación de los caminos

Los sucesivos vértices que conforman el camino mínimo se obtienen *volviendo hacia atrás*; es decir, desde el *último* que hizo que el camino fuera mínimo, al *último del último* y así sucesivamente hasta llegar al vértice origen del camino. Las llamadas recursivas del método (de la clase CaminoMinimo) que se escribe a continuación permiten, fácilmente, volver atrás y reconstruir el camino.

```
public void recuperaCamino(int v)
{
   int anterior = ultimo[v];
   if (v != s)
   {
      recuperaCamino(anterior); // vuelve al último del último
```

```
      System.out.print(" -> V" + v);
   }
   else
      System.out.print("V" + s);
   }
```

16.4. TODOS LOS CAMINOS MÍNIMOS: ALGORITMO DE FLOYD

En algunas aplicaciones resulta interesante determinar el camino mínimo entre todos los pares de vértices de un grafo dirigido y valorado. Considerando como vértice origen cada uno de los vértices del grafo, el algoritmo de Dijkstra resuelve el problema. Existe otra alternativa, más elegante y más directa, propuesta por Floyd que recibe el nombre de algoritmo de Floyd.

El grafo, de nuevo, está representado por la matriz de pesos, de tal forma que todo arco (v_i,v_j) tiene asociado un peso c_{ij}; si no existe arco, $c_{ij} = \infty$. Además, cada elemento de la diagonal, c_{ii}, se hace igual a 0. El algoritmo de Floyd determina una nueva matriz, D, de n x n elementos tal que cada elemento, D_{ij}, contiene el coste del camino mínimo de v_i a v_j .

El algoritmo tiene una estructura similar al algoritmo de Warshall para encontrar la matriz de caminos. Se generan consecutivamente las matrices D_1, D_2, ..., D_k, ..., D_n a partir de la matriz D_o que es la matriz de pesos. En cada paso se incorpora un nuevo vértice y se estudia si con ese vértice se puede mejorar los caminos para ser más cortos. El significado de cada matriz es:

D_0[i,j] = c_{ij} *coste* (peso) del arco del vértice i al vértice j

∞ si no hay arco.

D_1[i,j] = minimo(D_0[i,j],D_0[i,1]+D_0[1,j]).
Menor de los costes entre el anterior camino mínimo de i a j y la suma de costes de caminos de i a 1 y de 1 a j.

D_2[i,j] = minimo(D_1[i,j],D_1[i,2]+D_1[2,j]).
Menor de los costes entre el anterior camino mínimo de i a j y la suma de los costes de caminos de i a 2 y de 2 a j.

En cada paso se incorpora un nuevo vértice para determinar si hace el camino menor entre un par de vértices.

D_k[i,j] = minimo(D_{k-1}[i,j],D_{k-1}[i,k]+D_{k-1}[k,j]).
Menor de los costes entre el anterior camino mínimo de i a j y la suma de los costes de caminos de i a k y de k a j.

De forma recurrente, se añade en cada paso un nuevo vértice para determinar si se consigue un nuevo camino mínimo, hasta llegar al último vértice y obtener la matriz D_n, que es la matriz de caminos mínimos del grafo.

Ejemplo 16.2

La Figura 16.5 muestra un grafo dirigido con factor de peso y la correspondiente matriz de pesos. Aplicar el algoritmo de Floyd para obtener, en los sucesivos pasos, la matriz de caminos mínimos.

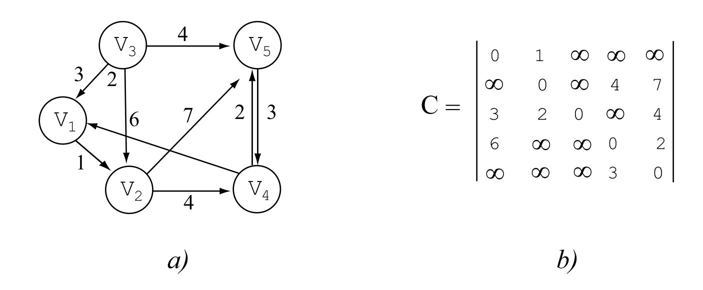


Figura 16.5 *a)* Grafo dirigido valorado de 5 vértices; *b)* matriz de pesos del grafo

El grafo consta de cinco vértices; por ello se forman cinco matrices: D_1, D_2, D_3, D_4, D_5 que es la matriz de caminos mínimos. La matriz D_0 es la matriz de pesos `C`.

$$D_2 = \begin{vmatrix} 0 & 1 & \infty & \infty & \infty \\ \infty & 0 & \infty & 4 & 7 \\ 3 & 2 & 0 & \infty & 4 \\ 6 & 7 & \infty & 0 & 2 \\ \infty & \infty & \infty & 3 & 0 \end{vmatrix}$$

Al incorporar el vértice `1` ha cambiado `D1(4,2)` ya que `C(4,1) + C(1,2) < C(4,2)`.

$$D_2 = \begin{vmatrix} 0 & 1 & \infty & 5 & 8 \\ \infty & 0 & \infty & 4 & 7 \\ 3 & 2 & 0 & 6 & 4 \\ 6 & 7 & \infty & 0 & 2 \\ \infty & \infty & \infty & 3 & 0 \end{vmatrix}$$

Al incorporar el vértice `2` ha habido varios cambios; así `D1(1,4): D1(1,2)+ D1(2,4)< D1(1,4)`. También cambian `D1(1,5)`, `D1(2,5)` y `D1(3,4)`.

$$D_3 = \begin{vmatrix} 0 & 1 & \infty & 5 & 8 \\ \infty & 0 & \infty & 4 & 7 \\ 3 & 2 & 0 & 6 & 4 \\ 6 & 7 & \infty & 0 & 2 \\ \infty & \infty & \infty & 3 & 0 \end{vmatrix}$$

Al incorporar el vértice `3` no hay cambios; al vértice `3` no llega ningún arco.

$$D_4 = \begin{vmatrix} 0 & 1 & \infty & 5 & 7 \\ 10 & 0 & \infty & 4 & 7 \\ 3 & 2 & 0 & 6 & 4 \\ 6 & 7 & \infty & 0 & 2 \\ 9 & 10 & \infty & 3 & 0 \end{vmatrix}$$

Al incorporar el vértice `4` cambian los elementos: `D3(1,5)`, `D3(2,1)`, `D3(5,1)` y `D3(5,2)`.

$$D_5 = \begin{vmatrix} 0 & 1 & \infty & 5 & 7 \\ 10 & 0 & \infty & 4 & 7 \\ 3 & 2 & 0 & 6 & 4 \\ 6 & 7 & \infty & 0 & 2 \\ 9 & 10 & \infty & 3 & 0 \end{vmatrix}$$

La incorporación del vértice `5` no genera cambios.

La matriz D_5 es la matriz de caminos mínimos. Dado cualquier par de vértices, se puede conocer la longitud del camino mas corto que hay entre ellos.

A tener en cuenta

Tanto el algoritmo de Dijkstra como el de Floyd permiten obtener los caminos mínimos, pero sólo se pueden aplicar en grafos valorados con *factor de peso* positivo, que son los mas frecuentes.

16.4.1. Codificación del algoritmo de Floyd

Se define la clase `TodoCaminoMinimo` para implentar el algoritmo al que se añade un pequeño cambio: guardar, en la matriz `traza`, el índice del último vértice que ha hecho que el camino sea mínimo desde el vértice v_i al v_j, para cada par de vértices del grafo.

```
package Grafo;

public class TodoCaminoMinimo
{
   private int [][] pesos;
   private int [][] traza;
   private int [][] d;
   private int n;          // número de vértices
   public TodoCaminoMinimo(GrafMatPeso gp)
   {
      n = gp.numeroDeVertices();
      pesos = gp.matPeso;
      d = new int [n][n];
      traza = new int [n][n];
   }
   public void todosCaminosMinimo()
   {
      // matriz inicial es la de pesos.
      for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++)
      {
         d[i][j] = pesos[i][j];
         traza[i][j] = -1; // indica que camino mas corto es el arco
      }
         //   Camino   mínimo   de   un   vértice   a   si   mismo:   0
      for (int i = 0; i < n; i++)
         d[i][i] = 0;
      for (int k = 0; k < n; k++)
         for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
               if ((d[i][k] + d[k][j]) < d[i][j])  // nuevo mínimo
               {
                  d[i][j] = d[i][k] + d[k][j];
                  traza[i][j]= k;
               }
   }
}
```

Eficiencia del algoritmo

El tiempo de ejecución del algoritmo crece rápidamente para grafos de relativamente muchos vértices, la complejidad es $O(n^3)$.

Para obtener la sucesión de vértices que forman el camino mínimo entre dos vértices se escribe el método `public void recuperaCamino(int vi, int vj)`, de igual forma que el método escrito en la clase `CaminoMinimo`, con la diferencia que ahora se utiliza la matriz `traza` en lugar de `ultimo`.

16.5. ÁRBOL DE EXPANSIÓN DE COSTE MÍNIMO

Los grafos no dirigidos se emplean para modelar relaciones simétricas entre *entes*, los vértices del grafo. Cualquier arista `(v,w)` de un grafo no dirigido está formada por un par no ordenado de vértices. Como consecuencia directa, la representación de un grafo no dirigido da lugar a matrices simétricas.

Una propiedad, que normalmente interesa conocer, de un grafo no dirigido es si para todo par de vértices existe un camino que los une; en definitiva, si el grafo es *conexo*. A los grafos conexos también se les denomina *Red Conectada*. El problema del *árbol de expansión de coste mínimo* consiste en buscar un árbol que *abarque* todos los vértices del grafo, con suma de pesos de aristas mínimo. Los árboles de expansión se aplican en el diseño de redes de comunicación.

Un *árbol*, en una red, es un subconjunto `G'` del grafo `G` que es conectado y sin ciclos. Los árboles tienen dos propiedades importantes:

1. Todo árbol de `n` vértices contiene exactamente `n-1` aristas.
2. Si se añade una arista a un árbol, se obtiene un ciclo.

> **Definición**
>
> *Árbol de expansión de coste mínimo*: es un subconjunto del grafo que abarca todos los vértices que están conectados cuyas aristas tienen una suma de pesos mínima.

Buscar un árbol de expansión de un grafo, en una red, es una forma de averiguar si está conectado. Todos los vértices del grafo tienen que estar en el árbol de expansión para que sea un grafo conectado. La Figura 16.6 muestra un grafo no dirigido y su árbol de expansión; este grafo es una *red conectada*.

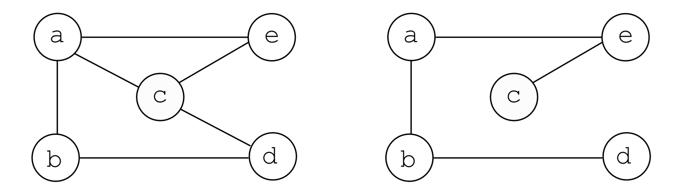


Figura 16.6 Grafo no dirigido y su árbol de expansión

El planteamiento del problema es el siguiente: dado un grafo no dirigido ponderado y conexo, encontrar el subconjunto del grafo compuesto por todos los vértices, con conexión entre cada par de vértices, sin ciclos y que cumpla que la suma de los pesos de las aristas sea mínimo. Estas operaciones encuentran el *árbol de expansión de coste mínimo*.

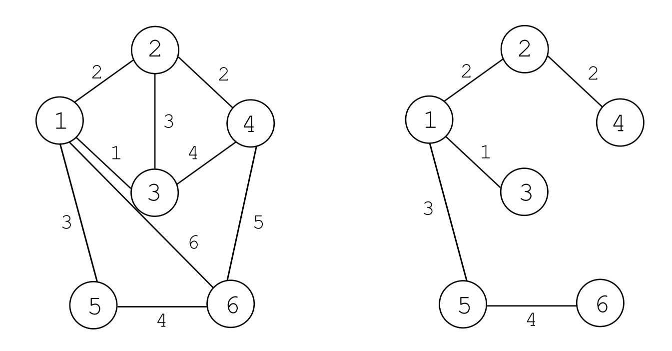


Figura 16.7 Grafo valorado y su árbol de expansión mínimo

16.5.1. Algoritmo de Prim

El algoritmo de Prim encuentra el árbol de expansión mínimo de un grafo no dirigido. Realiza sucesivos pasos, siguiendo la metodología clásica de los algoritmos *voraces:* hacer en cada paso *lo mejor que se pueda hacer*. En este problema, *lo mejor* consiste en incorporar al árbol una nueva arista del grafo de menor longitud.

Se parte de un grafo `G = (V,A)` no dirigido conectado, una red, donde `c(i,j)` es el *peso* o *coste* asociado al arco (v_i, v_j). Para describir el algoritmo, se suponen los vértices numerados de `1` a `n`. El conjunto `W` contiene los vértices que ya han pasado a formar parte del *árbol de expansión*.

El algoritmo arranca asignando un vértice inicial al conjunto `W`; por ejemplo el vértice `1`: `W = {1}`. A partir del vértice inicial, el *árbol de expansión* crece, añadiendo a `W`, en cada pasada otro vértice `z` todavía no incluido en `W`, de tal forma que si `u` es un vértice cualquiera de `W`, la arista `(u,z)` es la *más corta*, la de *menor coste*. El proceso termina cuando todos los vértices del grafo están en `W`, y por consiguiente, el *árbol de expansión* con todos los vértices está formado; además es mínimo porque en cada pasada se ha añadido la menor arista.

La Figura 16.8 muestra un grafo conectado, formado por `10` vértices. Inicialmente se incorpora al conjunto `W` el vértice `1`, `W = {1}`. De todas las aristas que forma parte el vértice `1`, la de menor coste es `(1,2)`, por esta razón se añade a `W` el vértice `2`, `W = {1, 2}`. La siguiente arista mínima, formada por cualquier vértice de `W` y los vértices no incorporados es `(2,4)`; entonces se añade a `W` el vértice `4`, `W = {1, 2, 4}`. La siguiente pasada añade el vértice `5`, al ser la arista `(4,5)` la mínima. Una nueva pasada incorpora el vértice `8` ya que la arista `(5,8)` es la de menor coste, `W = {1, 2, 4, 5, 8}`. A continuación se incorpora el vértice `7` ya que la arista `(5,7)` es la de menor coste, `W = {1, 2, 4, 5, 8, 7}`; le siguen los vértices `9, 10, 3` y termina el algoritmo con el vértice `6`, `W = {1, 2, 4, 5, 8, 7, 9, 10, 3, 6}`. La Figura 16.9 muestra el árbol de expansión de coste mínimo que se ha formado.

> ***Observación***
>
> En el sucesivos pasos del algoritmo de Prim, los vértices de `W` forman una componente conexa sin ciclos ya que las aristas elegidas tienen un vértice en `W` y el otro en los restantes vértices, `V-W`.

Otra forma de expresar el algoritmo de Prim: partiendo de un vértice inicial `u` se toma la arista menor `(u,v)` que no forme un ciclo, y de forma iterativa se toman nuevas aristas de menor peso `(z,w)`, sin dar lugar a ciclos y formando, en todo momento, una componente conexa.

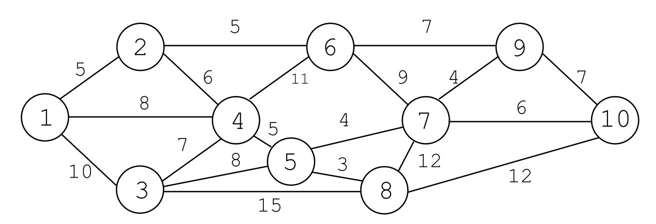


Figura 16.8 Grafo valorado conexo, representa una red telefónica

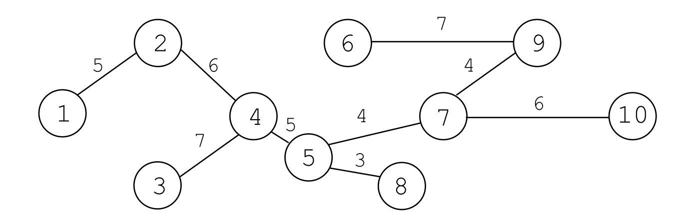


Figura 16.9 Árbol de expansión del grafo de la Figura 16.8

El algoritmo puede expresarse de la siguiente forma:

```
arbolExpansion Prim(G, T)
    G : grafo
    T : conjunto de arcos del árbol de coste mínimo
   var locales
    W: < conjunto de vértices >;
    u,w: vértices;
 inicio
    T <- {}
    V <- {1..n}
    u <- 1
    W <- {u}
    mientras W<> V hacer
      <Encontrar v de V-W tal que (u,v) sea mínimo>
      W <- W+{v}
      T <- T+{(u,v)}
    fin _ mientras
fin _ arbolExpansion
```

16.5.2. Codificación del algoritmo de Prim

La clase `ArbolExpansionMinimo` implementa los diversos algoritmos que calculan el árbol de expansión. El constructor inicializa la matriz de *costes* y el número de vértices.

Para resolver el problema de encontrar, en cada pasada, la arista de menor *peso* que une un vértice de `W` con otro de `V-W` se utilizan dos *arrays:*

- `masCerca`, tal que `masCerca[i]` contiene el vértice de `W` de menor *coste* respecto el vértice `i` de `V-W`.
- `coste`, tal que `coste[i]` contiene el peso de la arista `(i, masCerca[i])`.

En cada pasada se revisa `coste` con el fin de encontrar el vértice `z` de `V-W` que es el más cercano a `W`. A continuación se actualizan `masCerca` y `coste` teniendo en cuenta que `z` ha sido añadido a `W`.

```
package Grafo;

public class ArbolExpansionMinimo
{
   private int [][] Pesos;
   private int n; // vértice origen y número de vértices

   public ArbolExpansionMinimo(GrafMatPeso gp) // constructor
   {
      n = gp.numeroDeVertices();
      Pesos = gp.matPeso;
   }

   public int arbolExpansionPrim()    // implementación del algoritmo
   {
      int longMin, menor;
      int z;
      int [] coste = new int [n];
      int [] masCerca = new int [n];
      boolean [] W = new boolean [n];

      for (int i = 0; i < n; i++)
         W[i] = false;   // conjunto vacío
      longMin = 0;
      W[0] = true; //se parte del vértice 0
      // inicialmente, coste[i] es la arista (0,i)

      for (int i = 1; i < n; i++)
      {
         coste[i] = Pesos[0][i];
         masCerca[i] = 0;
      }
      for (int i = 1; i < n; i++)
      {  // busca vértice z de V-W mas cercano,
         // de menor longitud de arista, a algún vértice de W
         menor = coste[1];
         z = 1;
         for (int j = 2; j < n; j++)
            if (coste[j] < menor)
            {
               menor = coste[j];
               z = j;
            }
         longMin += menor;
         // se escribe el arco incorporado al árbol de expansión
         System.out.println("V" + masCerca[z] + " -> " + "V" + z);
         W[z] = true; // vértice z se añade al conjunto W
         coste[z] = GrafMatPeso.INFINITO;
         // debido a la incorporación de z,
         // se ajusta coste[] para el resto de vértices
         for (int j = 1; j < n; j++)
            if ((Pesos[z][j] < coste[j]) && !W[j])
            {
               coste[j] = Pesos[z][j];
```

```
                masCerca[j] = z;
            }
        }
        return longMin;
    }
}
```

16.5.3. Algoritmo de Kruscal

Kruskal propone otra estrategia para encontrar el árbol de expansión de coste mínimo. El árbol se empieza a construir con todos los vértices del grafo G pero sin aristas; se puede afirmar que cada vértice es una componente conexa en sí misma. El algoritmo construye componentes conexas cada vez mayores examinando las aristas del grafo en orden creciente del *peso*. Si la arista conecta dos vértices que se encuentran en dos componentes conexas distintas, entonces se añade la arista al árbol de expansión T. En el proceso, se descartan las aristas que conectan dos vértices pertenecientes a la misma componente, ya que darían lugar a un ciclo si se añaden al árbol de expansión ya que están en la misma componente. Cuando todos los vértices están en un solo componente, T, éste es el árbol de expansión de coste mínimo del grafo G.

El algoritmo de Kruskal asegura que el árbol no tiene ciclos, ya que para añadir una arista, sus vértices deben estar en dos componentes distintas; además es de coste mínimo, ya que examina las aristas en orden creciente de sus pesos.

La Figura 16.7 muestra un grafo y, a continuación, se obtiene su árbol de expansión, aplicando este algoritmo. En primer lugar se obtiene la lista de aristas en orden creciente de sus pesos:

```
{(1,3), (1,2), (2,4), (1,5), (2,3), (3,4), (5,6), (4,5), (1,6)}
```

La primera arista que se toma es (1,3), como muestra la Figura 16.10a; a continuación, las aristas (1,2), (2,4) y (1,5), en las que se cumple que los vértices forman parte de dos componentes conexas diferentes, como se observa en la Figura 16.10b. La siguiente arista, (2,3) como sus vértices están en la misma componente, por tanto se descarta; la arista (3,4) por el mismo motivo se descarta. Por último, la arista (5,6), que tiene sus vértices en dos componentes diferentes, pasa a formar parte del árbol, y el algoritmo termina ya que se han alcanzado todos los vértices.

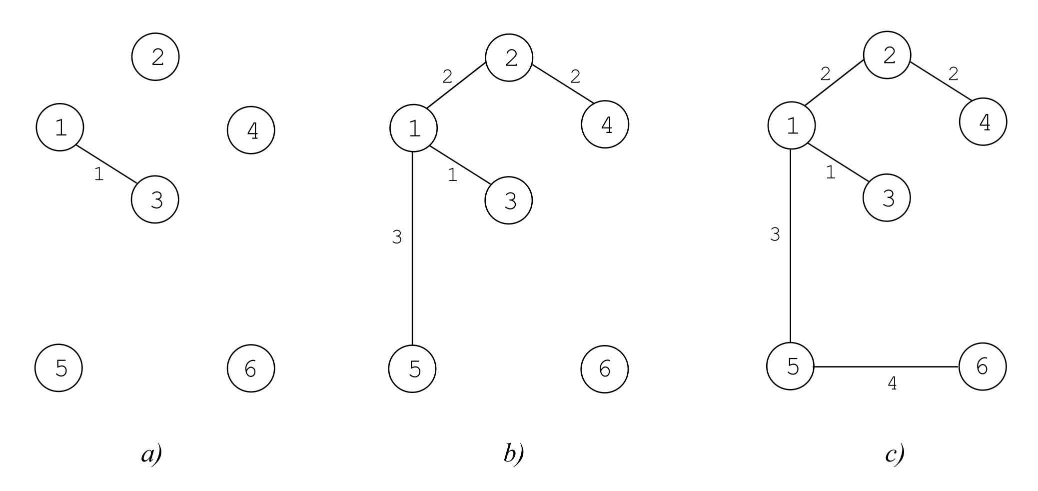


Figura 16.10 Formación del árbol de expansión del grafo de la Figura 16.7

RESUMEN

Si `G` es un *grafo dirigido sin ciclos*, entonces una ordenación topológica de `G` es una lista secuencial de los vértices de `G`, tal que para todos los vértices `v, w ∈ G` si existe una arista desde `v` a `w` entonces `v` precede a `w` en el listado secuencial. El término *acíclico* se utiliza con frecuencia para designar un grafo que no tiene ciclos.

La ordenación topológica es una forma de recorrer un grafo, pero sólo aplicable a grafos dirigidos acíclicos, que cumple la propiedad de que un vértice sólo se visita si ya han sido visitados todos sus predecesores. En un orden topológico, cada vértice debe aparecer antes que todos sus vértices sucesores en el *grafo dirigido*.

Una de las operaciones clave en un grafo es la búsqueda de *caminos mínimos*, es decir, caminos de menor longitud. La longitud de un camino se define como la suma de los *costes* de las aristas que lo componen. El *algoritmo de Dijkstra* determina el coste del camino mínimo desde un vértice al resto de vértices; es un algoritmo *voraz* que se aplica a un grafo valorado cuyas aristas tienen un *coste positivo.*

El *algoritmo de Warshall* es importante porque para todo par de vértices del grafo indica si hay un camino entre ellos. Con un estructura similar al anterior, el *algoritmo de Floyd* se utiliza para encontrar los caminos mínimos entre todos los pares de vértices de un grafo; también da la posibilidad de obtener la traza (secuencia de vértices) de esos caminos.

Un *árbol de expansión* es un árbol que contiene todos los vértices de una red. En un árbol no hay ciclos; es una forma de averiguar si la red está conectada. Uno de los problemas más comunes que se plantean sobre las redes es encontrar aquel *árbol de coste mínimo*. El *algoritmo de Prim* y el *algoritmo de Kruskal* resuelven eficientemente el problema de encontrar el árbol de expansión mínimo.

EJERCICIOS

16.1. Dada la red de la Figura 16.11 encontrar una ordenación topológica.

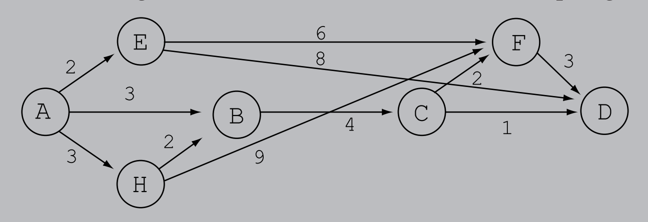


Figura 16.11 Grafo dirigido

16.2. En la red de la Figura 16.11 los arcos representan *actividades* y el *factor de peso* el tiempo necesario para realizar dicha actividad (un diagrama *Pert*). Cada vértice `v` de la red representa el tiempo que tardan todas las actividades que terminan en `v`. El ejercicio consiste en asignar a cada vértice `v` de la red el tiempo necesario para que todas las actividades que terminan en `v` se puedan realizar, esta característica se denomina `tn(v)`. Una estrategia que se puede seguir es asignar tiempo `0` a los vértices sin predecesores; si a todos los predecesores de `v` se les ha asignado tiempo, entonces

`tn(v)` es el máximo para cada predecesor, de la suma de tiempo del predecesor con el *factor de peso* del arco desde ese predecesor hasta `v`.

16.3. Considerando de nuevo la red de la Figura 16.11, y teniendo en cuenta el tiempo de cada vértice, `tn(v)`, calculado en el ejercicio 16.2, se quiere calcular el tiempo límite en el que todas las actividades que terminan en `v` pueden ser completadas, sin atrasar la terminación de todas las actividades, a este tiempo se le denomina `tl(v)`. La estrategia a seguir para este cálculo consiste en asignar `tn(v)` a todos los vértices `v` sin sucesores, si todos los sucesores de `v` tienen tiempo asignado, entonces `tl(v)` es el mínimo, entre todos los sucesores, de la diferencia entre el tiempo asignado al sucesor, `tl(v')`, y el factor de peso desde `v` hasta el sucesor `v'`.

16.4. Una ruta crítica de una red es un camino desde un vértice que no tiene predecesores hasta otro vértice que no tiene sucesores, tal que para todo vértice `v` del camino se cumple que `tn(v) = tl(v)`. Encontrar las rutas críticas de la red de la Figura 16.11.

16.5. La Figura 16.12 muestra una red conectada. Encontrar y dibujar un árbol de expansión de coste mínimo aplicando los pasos que propone el *algoritmo de Prim*.

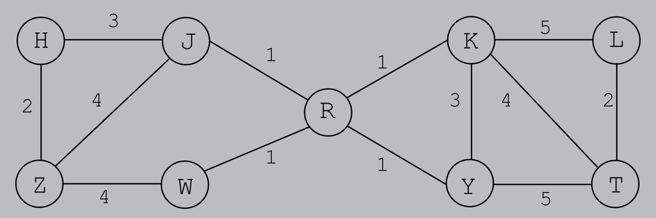


Figura 16.12 Red conectada

16.6. Encontrar el árbol de expansión de coste mínimo de la red conectada de la Figura 16.12, siguiendo los pasos del *algoritmo de Kruscal*.

16.7. El grafo de la Figura 16.13 es dirigido y las aristas tienen asociado un coste. Determinar el camino más corto desde el vértice `A` al resto de vértices del grafo.

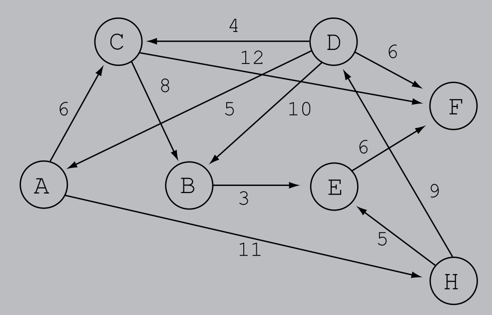


Figura 16.13 Grafo dirigido con factor de peso

16.8. Dado el grafo de la Figura 16.13, encontrar el camino más corto desde el vértice `D` al resto de vértices. Se han de seguir los pasos del *algoritmo de Dijkstra*, e incluir la secuencia de vértices que forman el camino.

16.9. En el grafo de la Figura 16.13, obtener los caminos más cortos entre todos los pares de vértices. Aplicar, paso a paso, el *algoritmo de Floyd*.

16.10. Dibujar un grafo dirigido con factor de peso en el que algún arco tenga factor de peso negativo. Al aplicar el *algoritmo de Dijkstra* desde un vértice origen, se ha de obtener algún resultado erróneo.

16.11. Escribir las modificaciones necesarias para que, teniendo como base el *algoritmo de Dijkstra*, se calculen los caminos mínimos entre todos los pares de vértices del grafo. ¿Cuál es la eficiencia de este algoritmo?

16.12. Tanto el algoritmo de *Prim* como el de *Kruscal* resuelven el problema de determinar el árbol de expansión mínimo de una red conectada. ¿Se pueden aplicar estos algoritmos si alguna arista tiene factor de peso negativo?

16.13. Para resolver el problema de encontrar los caminos mínimos desde un vértice origen al resto de los vértices, con aristas de coste negativo, se puede aplicar el algoritmo propuesto por *Bellman-Ford*. Este algoritmo no resuelve el problema si el grafo tiene ciclos; en el caso de existir ciclos con aristas de coste negativo, el algoritmo lo indica y da por terminada la búsqueda. Dado un grafo de `n` vértices, considerando como origen el vértice `1` y donde `P` es la matriz de pesos, el algoritmo de *Bellman-Ford* en seudocódigo es:

```
inicio
     desde v ← 2 hasta n hacer
        d(v) ← ∞
     fin _ desde
     s ← 1         { vértice origen }
     d(s) ← 0
     desde i ← 1 hasta n - 1 hacer
        para cada arista (u,v) hacer
           si d(v) > d(u) + P(u,v) entonces
              d(v) ← d(u) + P(u,v)
           fin _ si
     fin _ desde

     { prueba de ciclos con alguna arista de peso negativo }

     para cada arista (u,v) hacer
           si d(v) > d(u) + P(u,v) entonces
              Error ← true
           fin _ si

{ en el array d están los caminos mínimos desde
  el vértice s al resto de vértices }

fin
```

Como ejercicio, se solicita dibujar dos grafos con alguna arista que tenga factor de peso negativo, y aplicar el algoritmo descrito. En un grafo hacer que exista un ciclo con una arista de peso negativo.

16.14. Determinar, en el grafo de la Figura 16.14, los vértices que son *puntos de articulación*.

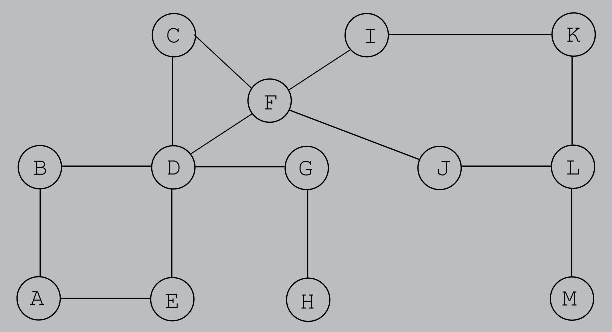


Figura 16.14 Grafo no dirigido con puntos de articulación

PROBLEMAS

16.1. Dado un grafo dirigido sin ciclos y el factor de peso que representa un *Pert*, escribir un programa que tenga como entrada la matriz de pesos y calcule:

a) El tiempo *tn(v)*.
b) El tiempo *tl(v)*.

La definición de estas magnitudes se encuentra en los ejercicios 16.2 y 16.4.

16.2. Un grafo que representa a una red es poco denso; en un programa se quiere implementar el grafo mediante *listas de adyacencia*. El programa tiene que encontrar las *rutas críticas* del grafo.

16.3. Se tiene una red (un *Pert*) representada con su matriz de pesos. Escribir un programa que calcule el mínimo tiempo en el que todo trabajo se puede terminar si tantas tareas como sea posible son realizadas en paralelo. El programa debe de escribir el tiempo en el que se inicia y se termina toda actividad en la red.

16.4. Escribir un programa tal que dada una red se determine el tiempo mínimo de realización del trabajo si como máximo se pueden realizar *n* actividades (de las posibles) en paralelo. El programa tiene como entrada la matriz de pesos del grafo; la salida del programa muestra el tiempo de inicio y el de finalización de cada actividad.

16.5. Cada iteración que realiza el algoritmo de *Dijkstra* selecciona aquel vértice para el cual el camino es mínimo desde el vértice origen. Implementar el algoritmo teniendo en cuenta que si en una iteración hay dos vértices en los que coincide la distancia mínima, seleccionar aquel con el menor número de arcos.

16.6. Tomando como base el algoritmo de *Dijkstra*, implementar un algoritmo que resuelva el problema siguiente: dado un grafo `G`, representado por su matriz de pesos, encuentre los caminos mínimos desde todo vértice `v` a un mismo vértice destino `d`.

16.7. Escribir un programa en el que dado un grafo *valorado*, con ciertos pesos de aristas negativos, determine los caminos mínimos desde un vértice origen al resto de los vértices. Utilizar el *algoritmo de Bellman-Ford* (Ejercicio 16.13).

16.8. Dado un grafo no dirigido con factor de peso, escribir un programa que tenga como entrada dicho grafo, lo represente en memoria y encuentre el árbol de expansión de coste mínimo.

16.9. Un *circuito de Euler* en un grafo dirigido es un ciclo en el cual toda arista es visitada exactamente una vez. Se puede demostrar que un grafo dirigido tiene un *circuito de Euler* si y sólo si es fuertemente conexo y para todo vértice el grado de entrada es igual al grado de salida. Escribir un programa que implemente un grafo dirigido mediante *listas de adyacencia* e implemente un algoritmo para encontrar, si existe, un *circuito de Euler*.

16.10. El algoritmo de *Warshall* determina la matriz de caminos de un grafo. La implementación con la matriz de adyacencia es adecuada en el caso de que el grafo sea lo suficientemente *denso*. Implementar el algoritmo para un grafo *poco denso*, representado con listas de adyacencia.

16.11. Se desea escolarizar una zona rural compuesta de 4 poblaciones, Centenera, Lupiana, Atanzón y Pinilla. Para ello se va a construir un único centro escolar en la población con el mejor coste del desplazamiento educativo, mínimo la función:

$z_i = \sum p_j \; d_{ij}$; donde p_j es la población escolar del pueblo `j` y d_{ij} es la distancia mínima del pueblo `j` al pueblo `i`.

Escribir un programa que tenga como entrada los datos de población escolar y distancia, en kilómetros, entre los pueblos. El programa determinará el pueblo donde conviene ubicar el centro escolar.

16.12. Un grafo `G` simula una red de dentros de distribución. Cada centro dispone de una serie de artículos y un *stock* de ellos, representado mediante una estructura lineal ordenada respecto al código de artículo. Entre los centros hay conexiones que no tienen por qué ser bidireccionales. Escribir un programa que represente el grafo dirigido ponderado (el coste de las aristas es la distancia, en kilómetros, entre los dos vértices). El programa tiene que resolver el problema siguiente: si un centro no tiene un artículo `z` y lo requiere, entonces el centro más cercano que disponga de ese dicho artículo `z` debe suministrarlo.

16.13. Europa consta de `n` capitales de estados, cada par de ellas está conectada, *o no,* por vía aérea. De estar conectadas (se supone por vuelo directo), la distancia en millas y el precio del vuelo se conoce. Las conexiones, aun siendo bidireccionales, no siempre tienen la misma distancia debido a las distintas rutas aéreas. Escribir un programa que se simule lo expuesto mediante un grafo, y se resuelva el problema siguiente: dada una cantidad de euros `E`, y un par de capitales `(c1, c2)`, averiguar la ruta más corta que se ajuste al dinero `E`.

16.14. Las ciudades dormitorio de una gran área metropolitana están conectadas a través de una red de carreteras, pasando por poblaciones intermedias, con el centro de la gran ciudad de dicha área metropolitana. Cada conexión entre dos nodos viene dada por el tiempo de desplazamiento, a la velocidad máxima de 50 km/h, entre los dos nodos. Escribir un programa que simule el supuesto mediante un grafo y determine el tiempo mínimo para ir desde una ciudad dormitorio al centro de la gran ciudad, y la sucesión de nodos por los que debe pasar.

CAPITULO 17
Colecciones

Objetivos

Con el estudio de este capítulo, usted podrá:

- Organizar grupos de objetos.
- Conocer las operaciones disponibles en las diversas clases *colección*.
- Crear un diccionario.
- Recorrer una colección mediante un iterador.

Contenido

Conceptos clave

- ♦ Contenedor.
- ♦ Diccionario.
- ♦ Genericidad.
- ♦ Iterador.
- ♦ `Object`.
- ♦ Tipo Abstracto de Datos.

INTRODUCCIÓN

Java dispone de un conjunto de clases para agrupar objetos, denominanadas colecciones. Cada clase organiza los objetos de un forma particular, como un mapa, una lista, un conjunto..., y se implementa como un *Tipo Abstracto de Datos*. Las colecciones son importantes para el desarrollo de programas profesionales dado que facilitan considerablemente el diseño y construcción de aplicaciones basadas en estructuras de datos tales como vectores, listas, conjuntos, mapas. La clase `Vector` procede de la versión original de Java. En *Java 2* se han potenciado las colecciones con nuevas clases e interfaces, todas ellas en el paquete `java.util`.

17.1. COLECCIONES EN JAVA

Las colecciones proporcionan programación genérica para muchas estructuras de datos. Una *colección* es una agrupación de objetos relacionados que forma una única entidad; por ejemplo, un *array* de objetos, un conjunto. El *array,* el vector, la matriz, en general, una *Colección,* es en sí misma otro objeto que se debe crear. Por ejemplo:

```
class Pueblo { .... };
class Puerto { ...  };

Pueblo [] col1 =  new Pueblo[100];
Puerto [] col2 = new Puerto [100];
LinkedList <String> conCad = // Lista de cadenas (en  Java 1.5)
new LinkedList <String>();
```

Las colecciones incluyen: *clases contenedoras* para almacenar objetos, *iteradores* para acceder a los objetos en el interior de los contenedores y *algoritmos* para manipular los objetos (métodos de clases).

Las clases *Colección* guardan objetos de cualquier tipo; de hecho, el elemento base es `Object` y, por consiguiente, debido a la conversión automática, se podrá añadir a la colección un objeto de cualquier tipo. El Ejemplo 17.1 crea una colección básica, secuencial, con un *array,* para guardar objetos de diferentes tipos.

Ejemplo 17.1

Considerar una aplicación en la que se debe almacenar `n` *objetos del tipo* `Punto2D`*,* `Punto3D` *y* `PuntoPolar`.

Se realiza un almacenamiento secuencial, para ello se declara una *array* de tipo `Object`. De esa forma se puede asignar cualquier objeto. El problema surge al recuperar los elementos del *array,* ya que se debe hacer un *cast* al tipo clase, que puede ser `Punto2D`, `Punto3D` y `PuntoPolar`.

```
class Punto2D { ... }
class Punto3D { ... }
class PuntoPolar { ... }
```

La declaración y creación del *array* de `N` elementos es:

```
final int N = 99;
Object [] rr = new Object[N];
int i = 0;
```

Asignación secuencial de objetos:

```
mas = true;
while (mas && (i < N))
{
   int opc;
   opc = menu();
   if (opc == 1)
      rr[i++] = new Punto2D();
   else if (opc == 2)
      rr[i++] = new Punto3D();
   if (opc == 3)
      rr[i++] = new PuntoPolar();
   else mas = false;
}
```

Asignar elementos en un *array* es una operación muy eficiente. Una de las limitaciones de los *arrays* es el tamaño, dado que al ser su tamaño fijo si se llena hay que ampliarlo. Por el contrario, una característica importante de las clases *Colección* es que se redimensionan automáticamente y, en consecuencia, el programador se despreocupa de controlar el número de elementos, y puede colocar tantos elementos como sea necesario.

> ***Norma***
>
> Cuando se vaya a trabajar con tipos de datos simples, o cuando se conozca el tipo de dato de los elementos y el tamaño final, resulta muy eficiente utilizar arrays. En cualquier otro caso, será conveniente utilizar alguna de las colecciones proporcionadas en `java.util`.

17.1.1. Tipos de Colecciones

En las primeras versiones del JDK, Java 1.0 y 1.1, se incorporaban colecciones básicas, aunque en la mayoría de las ocasiones suficientes. Las más importantes son `Vector`, `Stack`, `Dictionary`, `HashTable` y la interfaz `Enumeration` para recorrer los elementos de una colección.

Java 2 (versiones 1.2 y posteriores) incorpora nuevas clases de colección, dentro del paquete `java.util` (API de las colecciones). Existen tres tipos generales de colecciones: *conjuntos*, *listas* y *mapas*; los interfaces `Set`, `List` y `Map` describen las características generales de éstos. Además, la interfaz `Collection` especifica el comportamiento común de las colecciones.

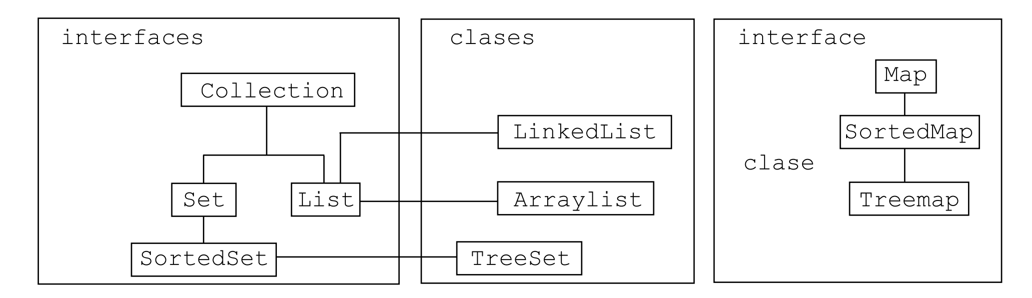


Figura 17.1 Algunos componentes de la jerarquía de colecciones

El diseño de estas colecciones tuvo muy en cuenta los problemas de seguridad que pudieran surgir con la ejecución de múltiples tareas. De tal forma que si varios hilos acceden a una de estas estructuras, se garantiza que los accesos están sincronizados, es decir que hasta que un hilo no termina de procesar los elementos de una colección no hay otro hilo que procesa la misma colección. La parte negativa de la sincronización es la ralentización de la ejecución de la aplicación; además, aun cuando no sean necesarios los comportamientos sincronizados, no hay forma de desactivar las comprobaciones de sincronización.

La declaración de métodos de la clase `Stack`, todos ellos con el comportamiento de sincronización es:

```
public synchronized int size()
public synchronized boolean isEmpty()
public synchronized boolean contains(Object elemento)
public synchronized Object put(Object clave, Object valor)
public synchronized Object remove(Object clave)
```

Las nuevas clases *Colección*, añadidas en Java 2, se han diseñado sin el comportamiento de sincronización, al menos directamente. Por ejemplo, obsérvese la siguiente declaración de métodos de la colección `ArrayList`:

```
public int size()
public boolean isEmpty()
public boolean contains(Object elemento)
public Object get(int index)
public Object set(int index, Object elemento)
public boolean add(Object elemento)
public Object remove(int index)
```

Ahora bien, hay aplicaciones en las que interesa proteger una colección de modificaciones simultáneas por más de un *hilo* `(Thread)`. Entonces, para añadir la cualidad de sincronización a las colecciones, Java 2 dispone de un mecanismo que dota de ese comportamiento a una colección. Estas características se implementan mediante métodos `static` de la clase `Collections`. Por ejemplo a:

```
ArrayList vd = new ArrayList();
```

se le dota de sincronización:

```
vd = Collections.synchronizedList(vd);
```

Para un mapa, el método `Collections.synchronizedMap()`; para un conjunto, `Collections.synchronizedSortedSet()`; y así para otras colecciones.

17.2. CLASES DE UTILIDADES: `Arrays` Y `collections`

La clase `Arrays` agrupa algoritmos útiles que se aplican, en general, a *arrays* de los tipos primitivos. `Collections` también es una clase de utilidades, de métodos `static` que implementan algoritmos aplicados a todo tipo de colecciones. `Vector`[1], *colección* histórica, está diseñada para guardar objetos de cualquier tipo; crece dinámicamente, sin necesidad de comprobar la capacidad y la propia clase se encarga de aumentar la capacidad si fuera necesario.

---

1 Véase el Capítulo 3.

17.2.1. Clase `Arrays`

Java incorpora la clase `Arrays` para disponer de métodos que trabajen con *arrays* (arreglos) de cualquier tipo. Estos métodos implementan algoritmos de búsqueda, ordenación y de asignación de un valor al *array* completo. Todos los métodos son `static` (métodos de clase). No se pueden crear objetos de la clase `Arrays` ya que su constructor es privado.

Ordenación de *arrays*

El método de ordenación, `sort()`, está sobrecargado, de tal forma que se puede ordenar un *array* de cualquier tipo primitivo y, en general, de tipo `Object`. `sort()` que implementa el algoritmo de ordenación *quicksort* que asegura una eficiencia *n log n*. Por ejemplo:

```
double [] w = {12.4, 5.6, 3.5, -2.0, 6.0, -4.5, 22.0};
// Llamada a sort() para ordenar w
Arrays.sort(w);
```

A continuación se muestran algunos de los métodos de ordenación:

```
public static void sort(double [] w);
public static void sort(int [] w);
public static void sort(long [] w);
public static void sort(Object [] w);
public static void sort(Object [] w, Comparator cmp);
```

Se puede ordenar un *subarray,* para ello se especifica el índice *inicio* (inclusive) y *final* (exclusive):

```
// ordena los elementos w[inicio] .. w[final-1]
public static void sort(double [] w, int inicio, int final );
public static void sort(int [] w, int inicio, int final);
...
```

> ***Norma***
>
> Para ordenar un array de objetos los elementos deben de implementar la interfaz `Comparable` ya que el criterio de ordenación está determinado por el método:
>
> `int compareTo(Object a);`

Ejemplo 17.2

Declarar la clase `Racional` *para representar un número racional (numerador, denominador) y llenar un array de objetos de esa clase. A continuación, ordenar de forma creciente el array de número racionales.*

La clase `Racional` tiene dos atributos de tipo `int`: numerador y denominador. Implementa la interfaz `Comparable` para poder realizar la ordenación con el método `Arrays.sort()`. Por ello es necesaro definir el método `int compareTo()` de tal forma que devuelva `-1, 0, 1` si el número racional del primer operando `(this)` es menor, igual o mayor, respectivamente que el número racional pasado como argumento.

```
import java.util.*;

class Racional implements Comparable
{
   private int numerador, denominador;
   public Racional()throws Exception
   {
     this(0,1);
   }
   public Racional(int n, int d) throws Exception
   {
      numerador = n;
      denominador = d;
      if (denominador == 0) throw new Exception("Denominador 0");
   }
   public String toString()
   {
      return numerador + "/" + denominador;
   }
   public int compareTo(Object x)     // método del interface
   {
      Racional r;
      r = (Racional) x;
      if (valorReal() < r.valorReal())
         return -1;
      else if (valorReal() > r.valorReal())
         return 1;
      else
         return 0;
   }
   private double valorReal()
   {
      return (double)numerador/(double)denominador;
   }
      // ...
}

// clase principal, crea los objeto de manera aleatoria,
// se escriben en pantalla, a continuación se ordena, y por último
// se vuelve a escribir.

public class OrdenaRacional
{
   static int MR = 7;
   public static void main(String[] a)
   {
      Racional [] ar = new Racional[MR];

      try {
         // numerador y denominador se genera aleatoriamente
         for (int i = 0; i < MR; i++)
         {
            int n, d;
            n = (int)(Math.random()* 21 +1);
            d = (int)(Math.random()* 21 +1);
            ar[i] = new Racional(n, d);
         }
```

```
      }
      catch (Exception e) {;}
      // listado de los objetos creados
      System.out.println(" Lista de numeros racionales: ");
      escribe(ar);
      // ordenación del array
      Arrays.sort(ar);
      // listado de los objetos ordenados
      System.out.println(" Lista ordenada de numeros racionales: ");

      escribe(ar);
   }

   static void escribe(Racional [] r)
   {
      for (int i = 0; i < r.length; i++)
         System.out.print(r[i] + "  ");
      System.out.println();
   }
}
```

Ejecución

```
Lista de numeros racionales:
11/5  2/17  13/12  1/19  13/19  3/11  7/19
Lista ordenada de numeros racionales:
1/19  2/17  3/11  7/19  13/19  13/12  11/5
```

Búsqueda de una clave

La operación de búsqueda se realiza sobre un *array* (arreglo) ordenado. La clase `Arrays` dispone del método `static binarySearch()` para realizar la búsqueda de un elemento en un *array*. El método devuelve la posición del elemento en el *array,* o bien, si no está, `-p` siendo `p` la posición de inserción del elemento en el *array*. El algoritmo que utiliza el método es el de búsqueda binaria que asegura una eficiencia de *log n*. El método está sobrecargado para cada tipo de dato primitivo y para *arrays* de cualquier objeto `(Object)` que implemente la interfaz `Comparable`. Por ejemplo,

```
int [] w = {14, -5, 3, 2, 6, -4, 22, 4};
   // llamada a sort() para ordenar w
Arrays.sort(w);
   // búsqueda de un elemento
int k;
k = Arrays.binarySearch(w, 4);
if (k >= 0)
   System.out.println("Posición de " + 4 + " en la lista: " + k);
```

La declaración de alguno de estos métodos es:

```
public static int binarySearch(double [] w, double clave);
public static int binarySearch(int [] w, int clave);
public static int binarySearch(Object [] w, Object clave);
public static int binarySearch(Object[]w, Object clave, Comparator c);
```

Asignación de un elemento

Otra utilidad de la clase `Arrays` es el método `fill()` para asignar un elemento a todas las posiciones de un *array,* o bien a un rango del *array.* El método está sobrecargado, y existe una declaración para cada tipo de dato primitivo y para `Object`. La declaración de `fill()` para algunos tipo es:

```
public static void fill (byte [] w, byte val)
public static void fill (byte [] w, int inicio, int final, byte val);
public static void fill (char [] w, char val)
public static void fill (char [] w, int inicio, int final, char val);
public static void fill (int  [] w, int val);
public static void fill (int  [] w, int inicio, int final, int val);
public static void fill (Object [] w, Object val);
...
```

Ejemplo 17.3

Definir un array de enteros, inicializar la primera mitad a `-1` *y la segunda mitad a* `+1`. *Además, inicializar un array de caracteres a la letra* `'a'` *y un array de cadenas a* `"Paloma"`.

Se definen *arrays* de los tipos solicitados y de tamaño constante. Para inicializar la primera mitad del *array* `a[]` es preciso especificar el índice `inicio = 0` y `final = a.length/2`; la segunda mitad tiene como `inicio` el anterior `final`.

```
static final int N = 10;
int [] iv = new int[N];
char [] cv = new char [N];
String [] sv = new String [N];
// llenado de los arrays
Arrays.fill(iv, 0, iv.length/2, -1);
Arrays.fill(iv, iv.length/2 +1, iv.length -1, 1);

Arrays.fill(cv, 'a');
Arrays.fill(sv, "Paloma");
```

17.2.2. Clase `Collections`

Esta clase se encuentra en el paquete `java.util` y está diseñada para trabajar con colecciones: `List`, `Map`, `Set`; en general sobre cualquier `Collection`. Agrupa métodos `static` que implementan algoritmos genéricos de ordenación, búsqueda, máximo y mínimo. Además, dispone de métodos para dotar de la características de sincronización a una colección, y para convertir una colección a *sólo lectura*.

Ordenación y búsqueda

Los métodos de ordenación se aplican a una lista cuyos elementos implementan la interfaz `Comparable` y permiten que se puedan comparar mutuamente. También, hay una sobrecarga de estos métodos para realizar la comparación con la interfaz `Comparator` .

```
public static void sort(list lista);
public static void sort(List lista, Comparator cmp);
public static int binarySearch(List lista, Object clave);
public static int binarySearch(List lista, Object cl, Comparator cmp);
```

Máximo y mínimo

Los métodos `max()` y `min()` devuelven el máximo y mínimo, respectivamente, de una colección. Para que se pueda realizar la operación, todos los elementos deben implementar la interfaz `Comparable` y ser mutuamente comparables. Es decir, si por ejemplo una colección guarda números complejos y cadenas, difícilmente se podrá obtener el máximo (además de ser absurdo). Los métodos son los siguientes:

```
public static Object max (Collection c);
public static Object min (Collection c);
// sobrecarga que obtiene el máximo o mínimo respecto a un comparador
public static Object max (Collection c, Comparator  cmp);
public static Object min (Collection c, Comparator  cmp);
```

Sincronización

Para añadir la cualidad de sincronización a las colecciones, `Collections` dispone de métodos que se aplican a cada tipo de colección:

```
public static Map synchronizedMap(Map mapa);
public static Set synchronizedSet(Set cn);
```

Conversión a sólo lectura

Estos métodos convierten la colección al modo sólo lectura, de tal forma que la operación de añadir `(add)` o eliminar `(remove)` un elemento levanta la excepción `UnsupportedOperationException`. Algunos de los métodos son los siguientes:

```
public static List unmodifiableList(List lista);
public static Set unmodifiableSet (Set conjunto);
public static Collection unmodifiableCollection (Collection c);
```

A continuación se escribe un ejemplo, en el que se crea un `Set` y después se convierte a modo *sólo lectura*.

```
Set  cn = new HashSet();
cn.add("Marta");
// crea una visión de cn no modificable, de sólo lectura
cn = Collections.unmodifiableSet(cn);
```

Utilidades

La clase `Collections` dispone de métodos útiles para ciertos procesos algorítmicos. El método `nCopies()` crea una lista con `n` copias de un elemento; el método `copy()` crea una lista copia de otra; `fill()` rellena todos los elementos de una lista con un objeto.

```
public static List nCopies (int n, Object ob);
public static void copy(List destino, List fuente);
public static void fill(List lista, Object ob);
```

El método `reverse()` invierte la posición de los elementos de una lista. Si la lista está en orden creciente, la llamada `Collections.reverse(lista)` deja la lista en orden decreciente.

```
public static void reverse(List lista);
```

Incluye, también el método `shuffle()` para reordenar aleatoriamente los elementos de una lista.

```
public static void shuffle(List lista);
```

Ejemplo 17.4

Se realizan diversas operaciones utilizando métodos de la clase `Collections`*: crear una lista a partir de la copia* `n` *veces de una cadena, ordenar una lista de objetos* `Integer`*, buscar el máximo, ...*

Se escribe el método `main()` con las operaciones `nCopies()`, `sort()`, `max()`, `min()` y `reverse()`. La declaración de la lista de objetos especifica que el tipo de los elementos es `Integer`, de esa forma el compilador realiza comprobaciones de tipo; también se puede realizar de forma genérica: `List lista = new ArrayList()`, de tal forma que se podría añadir cualquier tipo de objeto.

```
public static void main(String[] args)
{
   int n = 11;
   List lista1;
   // Crea una lista formada por n copias de "Marga"
   lista1 = Collections.nCopies(n, "Marga");
   System.out.println(lista1);
   // Crea una lista de objetos Integer, se ordena y se invierte
   List<Integer> lista2 = new ArrayList<Integer>();
   for (int i = 1; i <= n ; i++)
      lista2.add(new Integer((int)(Math.random()*100 +1)));
   System.out.println(lista2);
   System.out.println("Máximo: " + Collections.max(lista2)
                 + " \t Mínimo: " +  Collections.min(lista2));
   Collections.sort(lista2);
   System.out.println(lista2);
   Collections.reverse(lista2);
   System.out.println(lista2);
}
```

Ejecución

```
[Marga, Marga, Marga, Marga, Marga, Marga, Marga, Marga, Marga, Marga, Marga]
[26, 1, 97, 7, 46, 76, 98, 69, 53, 50, 76]
Máximo: 98    Mínimo: 1
[1, 7, 26, 46, 50, 53, 69, 76, 76, 97, 98]
[98, 97, 76, 76, 69, 53, 50, 46, 26, 7, 1]
```

17.3. COMPARACIÓN DE OBJETOS: `Comparable` Y `Comparator`

Numerosas operaciones con colecciones exigen que sus elementos sean comparables, es decir, que se pueda determinar si un elemento es menor, igual o mayor que otro. Esta propiedad se establece a nivel de clase, implementando la interfaz `Comparable`, o bien la interfaz `Comparator`.

17.3.1. Comparable

La interfaz `Comparable` se utiliza para establecer un orden natural entre los objetos de una misma clase. La declaración de la interfaz (paquete `java.lang`) es:

```
public interface Comparable
{
   public int compareTo(Object ob);
}
```

Si `compareTo()` devuelve un valor negativo significa que el objeto que llama al método es menor que el pasado en el argumento; si devuelve `0` significa que son iguales, y si devuelve un valor positivo el objeto que llama al método es mayor que el pasado en el argumento. La clase `Racional` del Ejemplo 17.2 implementa la interfaz `Comparable`; el programador de la clase ha fijado la forma de comparar números racionales.

> ***Nota***
>
> Todas las clases que representan a tipos primitivos (*wraper)*: `Integer`, `Double`, `...`, y la clase `String` implementan la interfaz `Comparable`. La implementación de `compareTo()` en `String` distingue mayúsculas de minúsculas.

17.3.2. Comparator

Hay métodos de ordenación y búsqueda de objetos que utilizan la interfaz `Comparator` para determinar el orden natural entre dos elementos. Su interfaz se encuentra en el paquete `java.util`. Su declaración es la siguiente:

```
public interface Comparator
{
   public int compare (Object ob1, Object ob2);
   public boolean equals(Object ob);
}
```

El método `compare()` relaciona dos objetos, no necesariamente del mismo tipo. Devolverá un valor negativo si el objeto `ob1` es menor que el segundo objeto, *cero* si son iguales, y valor positivo si `ob1` es mayor que `ob2`. El comportamiento de `compare()` tiene que ser compatible con el resultado del método `equals()`; es decir, si este devuelve `true`, `compare()` tendrá que devolver `0`. Por ejemplo:

```
class Cuadro
{
   int largo;
   int ancho;
   ...
}

class Compara implements Comparator
{
   public int compare (Object ob1, Object ob2)
   {
      Cuadro c1 = (Cuadro) ob1;
```

```
      Cuadro c2 = (Cuadro) ob2;
      if (c1.largo == c2.largo && c1.ancho == c2.ancho)
         return 0;
      else
      // c1 es menor que c2 si, en superficie, c1 es menor
         return (c1.largo * c1.ancho) - (c2.largo * c2.ancho);
   }
```

Se puede ordenar un *array,* o una lista, de *cuadros* pasando el objeto de *comparador* al método `sort()`:

```
Cuadro [] vcdos = new Cuadro[N];
Arrays.sort(vcdos, new Compara());
```

> ***Norma***
>
> Los métodos que utilizan un argumento de tipo `Comparator` se llaman pasando una referencia a la clase que implementa la interfaz `Comparator`.
>
> Por ejemplo:
>
> ```
> class Alba implements Comparator {...}
> Collections.sort(lista, new Alba());
> ```

17.4. Vector Y Stack

Tanto `Vector` como `Stack` (pila) son colecciones históricas para guardar objetos de cualquier tipo. Se puede colocar cualquier número de objetos ya que se redimensionan automáticamente.

17.4.1. Vector

El comportamiento de un vector se asemeja al de un *array,* con la característica de que no es necesario controlar su tamaño, ya que, si fuera necesario, aumentaría automáticamente dicho tamaño. Se pueden consultar los constructores de un `Vector`, los métodos para añadir un elemento y otros métodos de utilidad en el Capítulo 3. Además, a partir de Java 2 la clase `Vector` implementa la interfaz `List` para que forme parte de las colecciones de Java. La plataforma Java 5 permite, también, establecer el tipo concreto de elemento que puede guardar una colección, y en particular un vector, para realizar comprobaciones de tipo durante el proceso de compilación. Por ejemplo, un vector de cadenas `(String)`:

```
Vector<String> vc = new Vector<String>();
vc.addElement("Lontananza");
vc.addElement(new Integer(12));  // error de compilación
```

Sin embargo, si la declaración es la siguiente:

```
Vector vc = new Vector ();
```

se puede añadir cualquier tipo de elemento, no hay comprobación de tipo:

```
vc.addElement("Lontananza");
vc.addElement(new Integer(12));  // correcto
```

17.4.2. `Stack`

La clase *Stack* hereda el comportamiento de un vector y además define las operaciones del tipo abstracto *Pila* (*último en entrar primero en salir*). Todas las operaciones se realizan por un único punto, el final (*cabeza* o *top*) de la pila. La declaración de *Stack* es la siguiente:

```
public class Stack extends Vector
{
   ...
}
```

Los métodos definidos por la clase son:

`public Stack();`	Constructor, crea una pila vacía.
`public Object push(Object n);`	Añade el elemento `n`, devuelve `n`.
`public Object pop();`	Devuelve elemento *cabeza* y lo quita de la pila.
`public Object peek();`	Devuelve elemento *cabeza* sin quitarlo de la pila.
`public boolean empty();`	Devuelve `true` si la pila está vacía.

Los elementos que almacena una colección *stack* son de tipo genérico (tipo `Object`), esto hace necesario realizar conversión de tipo cuando se extraen. La plataforma Java 5 permite parametrizar el tipo de los elementos que guarda el *stack,* de tal forma que el compilador verifica el tipo.

Ejemplo 17.5

Analizar mediante una pila si una palabra o frase es palíndromo[2].

Se lee la palabra del teclado y a la vez cada carácter se guarda en una pila. Los elementos de la pila han de ser de tipo `Character`. Por consiguiente, su declaración es: `Stack<Character> pila`.

```
import java.util.*;
import java.io.*;

public class Palindromo
{
   public static void main(String[] args)
   {
      BufferedReader entrada = new BufferedReader( new
                                 InputStreamReader(System.in));
      Stack<Character> pila;
      String palabra;
      boolean pal;
      pila = new Stack<Character>();
      try {
         system.out.println("Palabra o frase: ");
         while ((palabra = entrada.readLine()) != null)
         {
            for (int i = 0; i < palabra.length(); i++)
            {
               pila.push(new Character(palabra.charAt(i)));
            }
            pal = true;
            int i = 0;
```

[2] Un palíndromo es una palabra que se lee igual de izquierda a derecha que de derecha a izquierda.

```
            while (pal && !pila.empty())
            {
               Character q;
               q = pila.pop();
               pal = q.charValue() ==  palabra.charAt(i++);
            }
            if (pal && pila.empty())
               System.out.println(palabra + " es un palíndromo");
            else
               System.out.println(palabra + " no es un palíndromo");
         }
      }
      catch (Exception e) {;}
   }
}
```

17.5. ITERADORES DE UNA COLECCIÓN

Un iterador permite acceder a cada elemento de una colección sin necesidad de tener que conocer la estructura de la misma. Históricamente, Java 1.0 incorporó el iterador `Enumeration`, posteriormente Java 2 desarrolla dos nuevos iteradores: `Iterador` y `ListIterator`.

17.5.1. Enumeration

La interfaz `Enumeration` declara métodos que recorren una colección. Este tipo de iterador permite acceder a cada elemento de una colección, pero al ser de sólo lectura no permite modificar la colección. `Enumeration` forma parte del paquete `java.util` y su declaración es la siguiente:

```
public interface Enumeration
{
   boolean hasMoreElements();
   Object nextElement();
}
```

`nextElement()` devuelve el siguiente elemento. Levanta la excepción `NoSuchElementException` si no hay mas elementos, es decir, si ya se ha recorrido toda la colección. La primera llamada devuelve el primer elemento.

`hasMoreElements()` devuelve `true` si no se ha accedido a todos los elementos de la colección. Normalmente se diseña un bucle, controlado por este método, para acceder a cada elemento de la colección.

Las colecciones históricas: `Vector`, `Stack`, `Dictionary`, `HashTable` disponen del método `elements()` que devuelve un tipo `Enumeration`, a partir del cual se puede recorrer la colección. Su declaración es:

```
Enumeration elements()
```

El esquema a seguir para acceder a cada elemento de una colección consta de los siguientes pasos:

1. Declarar una variable `Enumeration`.

```
Enumeration enumera;
```

2. Llamar al método `elements()` de la colección.

```
enumera = coleccion.elements()
```

3. Diseñar el bucle que obtiene y procesa cada elemento.

```
while (enumera.hasMoreElements())
{
   elemento = (TipoElemento) enumera.nextElement();
   <proceso de elemento>
}
```

Ejemplo 17.6

Crear una pila de diferentes objetos. Posteriormente, recorrer con un iterador `Enumeration` *y con métodos de* `Stack`.

La pila se llena de objetos `String`, `Integer` y `Double`, sin un orden establecido. Para recorrer la pila se crea un enumerador y un bucle hasta que no queden más elementos sin visitar. También se recorre aplicando la operación `pop()` y controlando que no esté vacía.

```
import java.util.*;
import java.io.*;

public class EnumeradorPila
{
  public static void main(String[] args)
  {
    final int N = 8;
    Stack pila = new Stack();
    String [] palabra =
              {"Urbion", "Magina", "Abantos", "Peralte", "Citores" };
    for (int i = 0; i < N; i++)
    {
      int n;
      n = (int)(Math.random()*N*2);
      if (n < palabra.length)
        pila.push(palabra[n]);
      else if (n < N+2)
        pila.push(new Double(Math.pow(n, 2)));
      else
        pila.push(new Integer(n * 3));
    }
      // crea un enumerador de la pila
    Enumeration enumera = pila.elements();
      // bucle para recorrer la pila
    System.out.println("Elementos de la pila " +
                 "en el orden establecido por el enumerador:");
    while (enumera.hasMoreElements())
    {
      Object q;
      q = enumera.nextElement();
      System.out.print(q + " ");
    }
```

```
        // bucle para recorrer la pila
      System.out.println("\nElementos de la pila en orden LIFO:");
      while (!pila.empty())
      {
        Object q;
        q = pila.pop();
        System.out.print(q + " ");
      }
    }
  }
```

Ejecución

```
Elementos de la pila en el orden establecido por el enumerador:
81.0 25.0 Urbion 49.0 36.0 36.0 64.0 64.0
Elementos de la pila en orden LIFO:
64.0 64.0 36.0 36.0 49.0 Urbion 25.0 81.0
```

17.5.2. Iterator

Java 2 desarrolla nuevas colecciones y el iterador común `Iterator`. Todo objeto colección se puede recorrer con este iterador. Todas las colecciones tienen el métodos `iterator()` que devuelve un objeto `Iterator`.

```
Iterator iter;
iter = Coleccion.iterator();
```

La interfaz `Iterator` permite no sólo acceder a los elementos, sino también eliminarlos. Pertenece al paquete `java.util`, y su declaración es:

```
public interface Iterator
{
   boolean hasNext();
   Object next();
   void remove();
}
```

`hasNext()` devuelve `true` si quedan elementos no visitados; es el equivalente a `hasMoreElements()`.

`next()` la primera llamada devuelve el primer elemento, según el orden establecido por el iterador.

`remove()` elimina de la colección el elemento obtenido por la última llamada a `next()`. Sólo se puede llamar una vez después de `next()`; en caso contrario, o bien si no ha habido una llamada a `next()`, levanta la excepción `IllegalStateException`. Normalmente, las colecciones implementan este método en un bloque sincronizado.

Norma

Se recomienda utilizar la interfaz `Iterator`; sustituye a `Enumeration` utilizada en colecciones históricas. Los métodos de acceso son más sencillos de recordar y además declara el método `remove()`.

Ejemplo 17.7

Dada una lista de puntos del plano se quiere eliminar de la lista aquellos cuya coordenada x esté fuera del rango [2, 12].

Se supone por declarada la clase `Punto` con el método `int getX()`, y la lista ya está creada. La lista es del tipo `ArrayList`. El fragmento de código escrito declara la lista y el iterador; realiza un bucle controlado por `hasNext()` para acceder a cada elemento y si la coordenada `x` no está en el rango `[2, 12]` lo elimina llamando al método `remove()`.

```
class Punto{...}

List lista = new ArrayList();
Iterator iter;
   // se llena la lista de objetos Punto
iter = lista.iterator();
while (iter.hasNext())
{
   Punto q;
   q = (Punto)iter.next();
   if (q.getX() < 2 || q.getX() > 12)
   {
      System.out.println("Punto: " + q + " se elimina");
      iter.remove();
   }
}
```

La colección guarda elementos de cualquier tipo `(Object)`, y ello exige realizar una conversión al tipo concreto `(Punto)` con el que se trabaja. Con Java 1.5 se puede parametrizar el tipo de los elementos, de tal forma que no es necesario realizar la conversión y, además, el compilador verifica el tipo de los elementos añadidos. A continuación se escribe el código con esta característica.

```
List<Punto> lista = new ArrayList<Punto>();
Iterator<Punto> iter;

iter = lista.iterator();
while (iter.hasNext())
{
  Punto q;
  q = iter.next();    // no es necesario un cast
  ....
```

17.5.3. ListIterator

Este iterador es específico de las colecciones que implementan la interfaz `List`. Permite recorrer una lista en ambas direcciones, y también eliminar, cambiar y añadir elementos de la lista. `ListIterator` deriva de `Iterator`, y su declaración es la siguiente:

```
public interface  ListIterator extends Iterator
{
   boolean hasNext();
   Object next();
   int nextIndex();
   boolean hasPrevious();
   Object previous();
```

```
    int previousIndex();
    void remove();
    void set(object):
    void add(Object);
}
```

Las colecciones con el comportamiento de `Lista` disponen de los métodos `listIterator()` y `listIterator(int i)` que devuelve un objeto `ListIterator`, de tal forma que la primera llamada a `next()` devuelve el primer elemento, o el de índice `i` respectivamente. Por ejemplo:

```
ListLinked ld = new ListLinked();
ListIterator iterBdir;

iterBdir = ld.listIerator();   //  el elemento actual es el primero
iterBdir = ld.listIterator(4); // elemento actual es el 4
```

Hay que tener en cuenta que el primer elemento es el de índice `0`, y que el previo a este tiene el índice `-1`.

El significado de `hasNext()` y `next()` es el mismo que tienen en la interfaz `Iterator`. Los métodos `hasPrevious()` y `previous()` permiten recorrer la lista en sentido inverso. `hasPrevious()` devuelve `true` si hay un elemento anterior. `previous()` devuelve el elemento anterior, levanta la excepción `NoSuchElementException` si no hay elemento previo.

`remove()` elimina el último elemento obtenido por `next()`, o bien por `previous()`.

`set(Object q)` sustituye por `q` el último elemento obtenido por `next()`, o bien por `previous()`.

`add(Object q)` inserta `q` inmediatamente antes del elemento que devolvería la llamada a `next()` y después del elemento que devolvería la llamada a `previous()`.

17.6. INTERFAZ `Collection`

La interfaz `Collection` describe el comportamiento común de las colecciones Java; es la raíz de la jerarquía de colecciones. En general, representa cualquier agrupación de objetos. Hay tres tipos de colecciones descritas por los interfaces `List`, `Set` y `Map`; las dos primeras derivan de `Collection`. La Figura 17.2 muestra la jerarquía de interfaces generales y la relación con los iteradores.

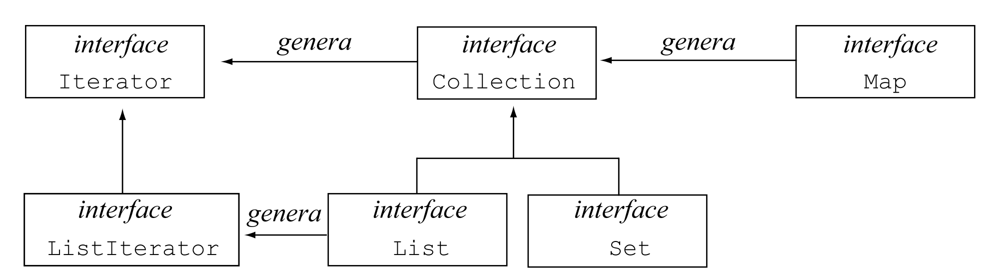


Figura 17.2 Jerarquía de interfaces

`Collection` declara métodos que serán implementados por las distintas clases, aunque muchos de ellos están especificados que son opcionales. Esto significa que la clase puede que no implemente el método con la funcionalidad especificada, sino que simplemente levante una excepción, en concreto la excepción `UnsupportedOperationException`.

Métodos para añadir

Los métodos para añadir elementos no están implementados en todas las colecciones. En el caso Java 2 son:

```
boolean add(Object ob);
boolean addAll(Collection c);
```

Devuelven `true` si la operación modifica la colección. Si el elemento que se inserta ya está en la colección, depende del tipo de ésta que se inserte o no.

Métodos para eliminar

Su implementación permite eliminar un elemento `(remove)`, o bien todos los que coincidan con los de la colección `(removeAll)` e incluso eliminar todos `(clear)`. Están especificados como opcionales y son los siguientes:

```
boolean remove(Object o);
boolean removeAll(Collection c);
void clear();
```

Además, el método `boolean retainAll(Collection c)` elimina los elementos que no estén en la colección `c`. Estos métodos son de tipo `boolean`, excepto `clear()`; devuelven `true` si la colección ha sido modificada, es decir, si ha habido alguna eliminación.

Métodos de búsqueda

Devuelven `true` si la colección contiene al elemento o elementos argumento.

```
boolean contains(Object o);
boolean containsAll(Collection c);
boolean equals(Object o);
```

El método `equals(Object o)` se recomienda implementar para comparar colecciones del mismo tipo; por ejemplo una lista con otra lista, y que devuelva `true` si son iguales en tamaño y en los elementos.

Métodos de colección

Todas las colecciones implementan el método `iterator()` que devuelve un objeto `Iterator` para recorrer la colección. El método `toArray()` devuelve un *array* con los elementos de la colección. A continuación se escribe la declaración de estos métodos y otros que se autoexplican.

```
Object[] toArray();
Iterator iterator();
boolean isEmpty();
int size();
```

17.7. LISTAS

Una lista es una agrupación lineal de elementos, que pueden duplicarse. A una lista se añaden elementos por la *cabeza*, por el *final* y, en general, por cualquier punto. También, se pueden eliminar elementos de uno en uno, o bien todos aquellos que estén en una colección. Existen dos tipos de listas: secuenciales y enlazadas. El concepto general de lista está representado por la interfaz `List`, esta interfaz es la raíz de la jerarquía y por conversión automática toda colección de tipo lista se puede tratar con una variable de tipo `List`. Por ejemplo:

```
List lista;
lista = new ArrayList();
lista = new LinkedList();
```

La jerarquía de listas se muestra en la Figura 17.3. Java 2 ha modificado las clases históricas `Vector` y `Stack` para ubicarlas en esta jerarquía; mantienen la funcionalidad histórica y, además, la funcionalidad heredada de la clase `AbstractList`.

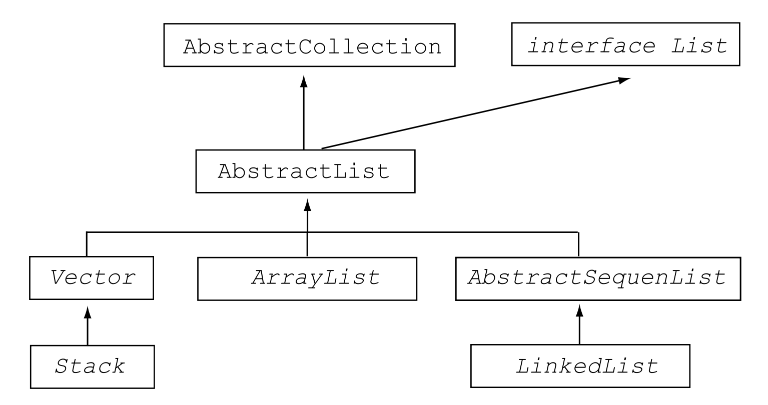


Figura 17.3 Jerarquía de colecciones `List`

`AbstractList` es una clase abstracta que se utiliza como esqueleto para implementar clases concretas con la característica de acceso aleatorio a los elementos, como un *array,* por lo cual se puede acceder a un elemento por un índice. Una clase que derive de ésta debe implementar los métodos `get(int indice)`, `size()`, y además, si va a realizar cambios, `set()`, `add()` y `remove()`; los métodos `iterator()`, `listIterator()`, `indexOf()` están definidos por esta clase. La clase `ArrayList` es una clase concreta que deriva de `AbstractList`; se utiliza para almacenar cualquier tipo de elementos, incluso está recomendado su uso en lugar de `Vector`.

`AbstractSequentialList` es la clase abstracta utilizada para implementar clases concretas de acceso secuencial; es decir, para clases que representen listas enlazadas. Clases concretas que deriven de `AbstractSequentialList` deben implementar los métodos `listIterator()`, `size()`, `add()`, `set()` y `remove()`. `LinkedList` es una clase concreta que permite agrupar objetos y organizarlos en forma de lista doblemente enlazada.

17.7.1. ArrayList

Esta clase agrupa elementos como un *array.* Es equivalente a `Vector`, pero con las mejoras introducidas por Java 2 se puede acceder a cualquier elemento, insertar o borrar a partir del índice en cualquier posición, aunque resulta un tanto ineficiente si se realiza en posiciones intermedias.

La clase `ArrayList` tiene tres constructores:

```
public ArrayList();
public ArrayList(int capacidad);
public ArrayList(Collection c);
```

Por ejemplo, se crea una colección con los elementos de un vector:

```
Vector v = new Vector();
ArrayList al = new ArrayList(v);
```

La clase `ArrayList` implementa los métodos de la interfaz `List`, y el método `clone()` de la interfaz `Cloneable` para poder crear una copia independiente de la colección.

Ejemplo 17.8

Se realizan las operaciones de añadir, eliminar, buscar y reemplazar con una colección de tipo ArrayList.

La colección va a estar formada por cadenas `(String)` leídas del teclado. Una vez formada la colección se elimina una cadena concreta y se reemplaza el elemento que ocupa la posición central. Para realizar una búsqueda se utiliza el método `indexOf()` que devuelve la posición que ocupa, o bien `-1`; a partir de esta posición se crea un iterador llamando al método `listIterator()` con el fin de recorrer y, a la vez, escribir los elementos.

```
import java.util.*;
import java.io.*;
public class ListaArray
{
   public static void main(String[] args)
   {
      BufferedReader entrada = new BufferedReader( new
                                        InputStreamReader(System.in));
      List av = new ArrayList();
      String cd;
      System.out.println("Datos de entrada (adios para acabar)");
      try {
         do {
            cd = entrada.readLine();
            if (! cd.equalsIgnoreCase("adios"))
               av.add(cd);
            else break;
         } while (true);

         System.out.println("Lista completa:" + av);
             // elimina una palabra
         System.out.println("Palabra a eliminar: ");
         cd = entrada.readLine();
         if (av.remove(cd))
            System.out.println("Palabra borrada, lista actual: + av");
         else
            System.out.println("No esta en la lista la palabra");
           // reemplaza elemento que está en el centro
         av.set(av.size()/2, "NuevaCadena");
         System.out.println("Lista completa:" + av);
           // busqueda de una palabra
         System.out.println("Búsqueda de una palabra: ");
         cd = entrada.readLine();
         int k = av.indexOf(cd);
          // crea iterador y recorre la lista hacia adelante
         if (k >= 0)
         {
            System.out.println("Recorre la lista a partir de k: "+k);
            ListIterator ls = av.listIterator(k);
            while(ls.hasNext())
```

```
            {
               System.out.print((String)ls.next() + " ");
            }
         }
      }
      catch(Exception e) {;}
   }
}
```

17.7.2. LinkedList

Esta clase organiza los elementos de una colección a la manera de una lista doblemente enlazada. Las operaciones de inserción y borrado en posiciones intermedias son muy eficientes; por el contrario, el acceso a un elemento por índice es ineficiente.

La clase dispone de un constructor sin argumentos que crea una lista vacía, y otro constructor que crea la lista con los elementos de otra colección.

```
public LinkedList();
public LinkedList(Collection c);
```

Implementa la interfaz `Cloneable`; las operaciones generales de las listas y métodos específicos que operan sobre el primer y último elemento son:

```
public Object getFirst()
public Object getLast()
public void addFirst(Object ob)
public void addLast(Object ob)
public Object removeFirst()
public Object removeFirst()
```

Se puede usar `LinkedList` para crear una estructura de datos *Cola* o *Pila*. Añadir datos a la *Cola* se hace con `addLast()`, eliminar con `removeFirst()` y obtener el elemento frente con `getFirst()`.

Ejemplo 17.9

Encontrar un numero capicúa leído del dispositivo estándar de entrada.

Se utiliza conjuntamente una *Cola* y una *Pila*. El número se procesa dígito a dígito (carácter del `'0'` al `'9'`), que se ponen en la cola y, a la vez, en la pila, pero como objeto de tipo `Character`. Para comprobar si es capicúa: se extraen consecutivamente elementos de la cola y de la pila, y se comparan por igualdad; si no se producen coincidencias es señal de que el número no es capicúa. El número es capicúa si el proceso de comprobación termina coincidiendo todos los dígitos en orden inverso, por consiguiente, tanto la pila como la cola quedan vacías. Tanto la pila como la cola son instancias de `LinkedList<Character>`.

```
import java.util.*;
import java.io.*;
class Cola
{
   private LinkedList<Character> qq;
```

```
   public Cola ()
   {
      qq = new LinkedList<Character> ();
   }
   public boolean colaVacia()
   {
      return qq.isEmpty();
   }
   public void insertar(Character elemento)
   {
      qq.addLast(elemento);
   }
   public Character quitar() throws Exception
   {
      return qq.removeFirst();
   }
   public void borrarCola()
   {
      qq.clear();
   }
   public Character frenteCola() throws Exception
   {
      return qq.getFirst();
   }
}

class Pila
{
   private LinkedList<Character> pila;
   public Pila()
   {
      pila = new LinkedList<Character> ();
   }
   public boolean pilaVacia()
   {
      return pila.isEmpty();
   }
   public void insertar(Character elemento)
   {
      pila.addFirst(elemento);
   }
   public Character quitar()throws Exception
   {
      return pila.removeFirst();
   }
   public void limpiarPila()
   {
      pila.clear();
   }
   public Character cimaPila() throws Exception
   {
      return pila.getFirst();
   }
}

public class capicuaCola
{
   public static void main(String[] args)
   {
      BufferedReader entrada = new BufferedReader( new
                                 InputStreamReader(System.in));
```

```
      String cd;
      try {
         do {
            System.out.println("Numero a probar si es capicua ");
            cd = entrada.readLine();
            if (! esCapicua(cd))
               System.out.println(cd + " No es capicua");
            else
            {
               System.out.println(cd +
                                 " es capicua, fin del programa");
               break;
            }
         } while (true);
      }
      catch(Exception e) {;}
   }

   static boolean esCapicua(String nm) throws Exception
   {
      Cola q = new Cola();
      Pila pila = new Pila();
      for (int i = 0; i < nm.length(); i++)
      {
         q.insertar(new Character(nm.charAt(i)));
         pila.insertar(new Character(nm.charAt(i)));
      }
      boolean es = true;
      while (es && !q.colaVacia())
      {
         Character c1, c2;
         c1 = q.quitar();
         c2 = pila.quitar();
         es = es && (c1.equals(c2));
      }
      return es;
   }
}
```

Ejecución

```
Numero a probar si es capicua
23456
23456 No es capicua
Numero a probar si es capicua
123321
123321 es capicua, fin del programa
```

17.8. CONJUNTOS

La estructura de datos `Conjunto` se basa en el concepto matemático de conjunto: colección de elementos no duplicados. Algebraicamente los elementos de un conjunto no tienen que mantener un orden; sin embargo, hay dos tipos de implementaciones, una, a partir de la interfaz `SortedSet`, mantiene en orden los elementos, otra sin un orden establecido. La interfaz `Set` declara las operaciones generales de los conjuntos y es la raíz de la jerarquía. Los métodos que declara son los mismos que `Collection`, aunque pone las restricciones que derivan de no tener elementos duplicados.

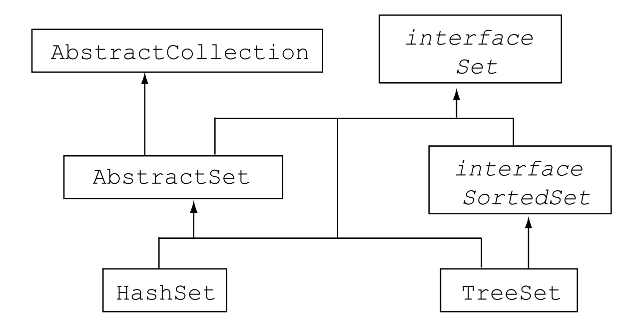


Figura 17.4 Jerarquía de colecciones `Set`

Todas las operaciones matemáticas de los conjuntos: *unión*, *intersección*, ..., se realizan con los métodos de las clases concretas `HashSet` y `TreeSet`.

17.8.1. AbstractSet

Esta clase abstracta deriva de `AbstractCollection` y por tanto hereda sus métodos tal como los define, que además no redefine. Sólo implementa los siguientes métodos:

```
public boolean equals(Object o)
public int hashCode()
public boolean removeAll(Collection c)
```

`equals()` devuelve `true` si el conjunto actual contiene los mismos elementos que el conjunto pasado en el argumento. Por ejemplo:

```
Set c1 = new TreeSet();
Set c2 = new HashSet();
if (c1.equals(c2)) ...
```

`removeAll(Collection c)` quita los elementos que están en la colección `c`. Devuelve `true` si ha habido alguna modificación o, lo que es igual, si al menos hay un elemento en `c`. Con este método se realiza la operación algebraica *diferencia* de conjuntos. Por ejemplo:

```
HashSet z = (HashSet) c2.clone();
z.removeAll(c1);
```

donde `z` es el conjunto diferencia `c2 - c1`.

> ***Norma***
>
> La clase de los elementos deben de implementar el método `equals()` para que tengan consistencia las comparaciones internas. Por ejemplo, para un conjunto de discos la clase `Disco`:
>
> ```
> class Disco
> {
>    public boolean equals(Object ob){
>       Disco d = (Disco) ob;
>       return (autor.equals(d.autor) && codeIsbn == d.getIsbn() ...}
> ```

17.8.2. HashSet

La clase `HashSet` guarda los elementos de un conjunto sin mantener un orden. Los elementos del conjunto se guardan en una tabla hash (`HashMap`). La declaración de la clase es la siguiente:

```
public class HashSet extends AbstractSet
             implements Set, Cloneable, Serializable
```

Utilizando los constructores de la clase `HashSet` se puede crear un conjunto vacío o un conjunto con los elementos de otra colección.

```
public HashSet();
public HashSet(Collection c);
```

Por ejemplo, se crea el conjunto `cnj` con los elementos no repetidos de lista:

```
List lista = ne LinkedList();
...
HashSet cnj = new HashSet(lista);
```

Los métodos que implementa `HashSet` son:

`public boolean add(Object ob)`	si el elemento no está lo añade y devuelve `true`.
`boolean remove(Object ob)`	si `ob` pertenece al conjunto es eliminado y devuelve `true`.
`public void clear()`	deja vacío el conjunto.
`public Iterator iterator()`	crea un iterador.
`public int size()`	devuelve el número de elementos.
`public boolean isEmpty()`	devuelve `true` si está vacío.
`public boolean contains(Object ob)`	devuelve `true` si `ob` pertenece al conjunto.
`public Object clone()`	crea una copia del conjunto.

Unión de conjuntos

La operación algebraica unión de dos conjuntos produce otro conjunto con los elementos comunes y no comunes. El método heredado de `AbstractCollection, addAll()`, realiza esta operación.

Intersección de conjuntos

La intersección de dos conjuntos produce otro conjunto con los elementos comunes. El método `retainAll()`, heredado de `AbstractCollection`, realiza esta operación.

Diferencia de conjuntos

La diferencia de dos conjuntos, `c2- c1`, es otro conjunto con los elementos de `c2` que no pertenecen a `c1`. La llamada a `removeAll()` produce el conjunto diferencia.

Ejemplo 17.10

Crear dos conjuntos de enteros y realizar las operaciones algebraicas: unión, diferencia e intersección.

Los conjuntos creados son instancias de la clase `HashSet`. Los elementos de los conjuntos se generan aleatoriamente. Como está redefinido el método `toString()`, para escribir los elementos de un conjunto, se ejecuta simplemente `System.out.println(`*conjunto*`)`. Las operaciones requeridas se realizan, respectivamente, con los métodos `addAll()`, `removeAll()` y `retainA-`

ll(). Con el fin de mostrar la utilización de un iterador, se recorre cada conjunto obtenido para mostrar sus elementos.

```
import java.util.*;
public class ConjHash
{
  public static void main(String[] args)
  {
     ConjHash a;
     HashSet cn2, cn1;
     a = new ConjHash();
     cn2 = a.creaConj();
     System.out.println("Conjunto cn2: " + cn2);
     cn1 = a.creaConj();
     System.out.println("Conjunto cn1: " + cn1);
       // union de conjuntos
     HashSet union;
     union = (HashSet)cn2.clone();
     union.addAll(cn1);
     System.out.print("cn2 + cn1: ");
     a.iteraConj(union);
        // diferencia de conjuntos
     HashSet dif;
     dif =(HashSet)cn2.clone();
     dif.removeAll(cn1);
     System.out.print("cn2 - cn1: ");
     a.iteraConj(dif);
        // intersección de conjuntos
     HashSet inter;
     inter = (HashSet)cn2.clone();
     inter.retainAll(cn1);
     System.out.print("cn2 * cn1: ");
     a.iteraConj(inter);
  }
  public void iteraConj(Set cnj)
  {
     Iterator ic;
     Integer q;
     ic = cnj.iterator();
     while (ic.hasNext())
     {
       q = (Integer) ic.next();
       System.out.print(q + " ");
     }
     System.out.println();
  }
  public HashSet creaConj()
  {
     HashSet q = new HashSet();
     int n = (int)(Math.random()*7 +3);
     for (int i = 0; i < n; i++)
     {
       boolean s;
       Integer e = (int)(Math.random()*17 +3);
       Integer r = new Integer(e);
       s = q.add(r);
       if (!s) System.out.println(e + " repetido");
     }
     return q;
  }
}
```

Ejecución

```
19 repetido
15 repetido
9 repetido
Conjunto cn2: [15, 4, 19, 9, 11, 12]
17 repetido
11 repetido
Conjunto cn1: [13, 4, 8, 11, 17, 5]
cn2 + cn1: 15 13 4 19 8 9 11 17 5 12
cn2 - cn1: 15 19 9 12
cn2 * cn1: 4 11
```

17.8.3. TreeSet

Los conjuntos de tipo `TreeSet` se diferencian de los `HashSet` por mantener en orden los elementos. La ordenación puede ser ascendente, es decir en el orden natural determinado por la interfaz `Comparable`; o bien el orden que establece la implementación de la interfaz `Comparator`. Los elementos del conjunto se organizan en un árbol *(árbol roji-negro)* que garantiza un tiempo de búsqueda de eficiencia logarítmica, `O(log n)`.

El comportamiento, es decir los métodos, de esta implementación se encuentran en la interfaz `SortedSet` que la clase `TreeSet` implementa. La declaración de la interfaz y de la clase son las siguientes:

```
public interface SortedSet extends Set

public class TreeSet extends AbstractSet
                     implements SortedSet, Cloneable, Serializable
```

La especificación de los métodos más importantes `SortedSet` está basada en la ordenación, son los siguientes:

`SortedSet subSet(Object` *desde*, `Object` *hasta*)	Devuelve el subconjunto formado por los elementos mayor o igual que, *desde* y menor que *hasta*.
`SortedSet tailSet(Object` *desde*)	Devuelve el subconjunto formado por elementos mayores o igual que *desde*.
`SortedSet headSet(Object` *hasta*)	Devuelve el subconjunto formado por los elementos estrictamente menores que *hasta*.

Los subconjuntos que se obtienen son "visiones" del conjunto origen; y toda modificación del subconjunto modifica el conjunto origen, y viceversa.

Además, `first()` y `last()` devuelven, respectivamente, el primero y último elementos.

Constructores

`TreeSet` dispone de cuatro constructores. El primero crea un conjunto vacío. El segundo también y además establece el objeto *comparator* a partir del cual realizará la ordenación de los elementos. Para que la ordenación sea posible, los elementos del conjunto deben tener la propiedad de relación según el objeto *comparator*.

```
public TreesSet();
public TreeSet(Comparator cmp);
```

Los otros dos constructores crean el conjunto a partir de otro conjunto, o con los elementos de cualquier colección.

```
public TreeSet(SortedSet conj);
public TreeSet(Collection c);
```

A continuación se crean conjuntos con estos constructores:

```
class Relacion implements Comparator { ... }
Relacion acm = new Relacion();   // objeto comparador
Set c1, c2, c3, c4;
List lista = new LinkedList();   // colección

c1 = new TreeSet();              // conjunto vacío
c2 = new TreeSet(acm);           // conjunto vacío

c3 = new TreeSet(c2);       // inicializado con conjunto ordenado
c4 = new TreeSet(lista);    // inicializado con una colección
```

Métodos implementados

`TreeSet` dispone de métodos heredados de la clase base `AbstractSet` e implementa los especificados en las interfaces de la declaración. Los métodos que añaden o eliminan elementos del conjunto mantienen la estructura ordenada y tienen la misma *signatura* (nombre y tipo) que los de la clase `HashSet`. Con estos métodos se realizan las operaciones básicas de conjuntos: unión, intersección y diferencia. El Ejemplo 17.11 realiza operaciones con un conjunto ordenado, crea dos conjuntos vacíos, uno de ellos con un objeto comparador que compara cadenas en orden alfabético ignorando el tipo de letra; es decir, sin distinguir mayúsculas de minúsculas.

> ***Nota de programación***
>
> La clase que define los elementos del conjunto debe implementar el método `equals()` para poder realizar búsquedas y comparaciones, tanto en los conjuntos `HashSet` como en los `TreeSet`.

Ejemplo 17.11

Crear dos conjuntos ordenados cuyos elementos sean cadenas de caracteres (String). Realizar operaciones de insertar, eliminar, obtener subconjunto y operaciones algebraicas de unión, diferencia e intersección.

Los elementos de los conjuntos son cadenas en un *array* (pueden cambiarse para que sean leídas de teclado). Con la finalidad de que la ordenación no tenga en cuenta mayúsculas o minúsculas, se declara la clase `Compara` para definir el método `compare()` de la interfaz `Comparator`. La definición convierte sus dos argumentos a cadenas con letras mayúsculas y las relaciona con el método `compareTo()`. Al ser todas las letras mayúsculas no hay posibilidad de distinguir, al comparar, mayúsculas de minúsculas.

Los dos conjuntos se crean vacíos, el segundo con el objeto comparador `Compara`. Posteriormente se realizan operaciones de añadir, obtener subconjuntos, insertar en el subconjunto, unión... Los conjuntos obtenidos se escriben en la pantalla para mostrar los resultados de las operaciones.

```
import java.util.*;

class Compara implements Comparator
{
  public int compare (Object x1, Object x2)
  {
    String c1 = (String) x1;
    String c2 = (String) x2;
    c1 = c1.toUpperCase();
    c2 = c2.toUpperCase();
    return c1.compareTo(c2);
  }
}

public class ConjOrdenado
{
    public static void main(String[] args)
    {

      String [] cad1 = {"Libro", "Mesa", "mes", "Papel", "Papelera",
                        "armario", "globo"};
      String [] cad2 = {"lupi", "Maria", "angela", "Julian",
                        "Esperanza", "Miguel", "maite", "marta"};

      TreeSet cor1, cor2;
      cor1 = new TreeSet();
      cor2 = new TreeSet(new Compara());

      for (int i = 0; i < cad1.length; i++)
        cor1.add(cad1[i]);
      for (int i = 0; i < cad2.length; i++)
        cor2.add(cad2[i]);
      System.out.println("Conjunto ordenado, sin comparador: "
                                                      + cor1);
      System.out.println("Conjunto ordenado, con comparador: "
                                                      + cor2);
      // eliminar un elemento
      if (cor1.remove("Mes"))
        System.out.println("Eliminado - Mes - de conjunto 1 "
                                                      + cor1);
      else
         System.out.println("No se encuentra - Mes - en conjunto 1 "
                                                      + cor1);

      if (cor2.remove("MAITE"))
        System.out.println("Eliminado - MAITE - de conjunto 2 "
                                                      + cor2);
      else
         System.out.println("No se encuentra - MAITE - en conjunto 2
                                                    " + cor2);
      // Obtener un subconjunto y operaciones
      Set sc1;
      sc1 = cor1.subSet("Libro", "armario");
      System.out.println("Subconjunto : " + sc1);
      sc1.add("Marta");
      System.out.println("Subconjunto modificado: " + sc1);
      System.out.println("Conjunto origen modificado: " + cor1);
      cor1.add("lupi"); cor1.add("Luna");
      System.out.println("Subconjunto modificado: " + sc1);
      System.out.println("Conjunto origen modificado: " + cor1);
```

```
        // Operaciones algebraicas: unión y diferencia
    TreeSet union = (TreeSet)cor2.clone();
    union.addAll(cor1);
    System.out.println("cor2 + cor1: " + union);
    TreeSet dif = (TreeSet)cor2.clone();
    dif.removeAll(cor1);
    System.out.println("cn2 - cn1: " + dif);

  }
}
```

Ejecución

```
Conjunto ordenado, sin comparador: [Libro, Mesa, Papel, Papelera, ar-
mario, globo, mes]
Conjunto ordenado, con comparador: [angela, Esperanza, Julian, lupi,
maite, Maria, marta, Miguel]
No se encuentra  - Mes - en conjunto 1 [Libro, Mesa, Papel, Papelera,
armario, globo, mes]
Eliminado - MAITE - de conjunto 2 [angela, Esperanza, Julian, lupi,
Maria, marta, Miguel]
Subconjunto : [Libro, Mesa, Papel, Papelera]
Subconjunto modificado: [Libro, Marta, Mesa, Papel, Papelera]
Conjunto origen modificado: [Libro, Marta, Mesa, Papel, Papelera, ar-
mario, globo, mes]
Subconjunto modificado: [Libro, Luna, Marta, Mesa, Papel, Papelera]
Conjunto origen modificado: [Libro, Luna, Marta, Mesa, Papel, Papelera,
armario, globo, lupi, mes]
cor2 + cor1: [angela, armario, Esperanza, globo, Julian, Libro, Luna,
lupi, Maria, marta, mes, Mesa, Miguel, Papel, Papelera]
cn2 - cn1: [angela, Esperanza, Julian, Maria, marta, Miguel]
```

17.9. MAPAS Y DICCIONARIOS

Un diccionario agrupa elementos identificados mediante claves únicas. En cierto modo, un diccionario es una colección cuyos elementos son pares y están formados por un dato y su clave, que identifica de manera unívoca al elemento. Por ejemplo, el *número de matricula* del conjunto de alumnos puede considerarse un campo clave para organizar la información relativa al alumnado de una universidad. Los diccionarios son contenedores asociativos, también denominados *mapas*.

Las colecciones históricas definen los diccionarios con la clase abstracta `Dictionary` y la clase concreta `HashTable`. La interfaz `Map` especifica los métodos comunes a los mapas de las colecciones desarrolladas en Java 2.

17.9.1. Dictionary

La clase abstracta `Dictionary` es la interfaz para crear diccionarios en las colecciones históricas, y procesa la colección como si fuera un *array* asociativo al que se accede por una clave. Tanto el campo dato como la clave de cada elemento del diccionario son objetos. Todos los métodos de la clase son abstractos (se debería haber declarado como interfaz):

```
abstract public boolean isEmpty();
abstract public int size();
abstract public Enumeration keys();
```
crea una enumeración para las claves.

```
abstract public Enumeration elements();        crea una enumeración para los valores.
abstract public Object get(Object clave);      devuelve valor asociado a clave.
abstract public Object put(Object clave, Object valor);      añade clave, valor.
abstract public Object remove(Object clave);   elimina el elemento de la clave.
```

Se especifica que `get()` devuelva `null` si no está la clave en el diccionario. El método `put()` no inserta el par `(clave, valor)` si la `clave` está en el diccionario, en cuyo caso devuelve el valor asociado. `remove()` devuelve `null` si la `clave` no se encuentra. Estos métodos levantan la excepción `NullPointerException` si la clave es `null`.

Se puede crear un tipo propio de diccionario extendiendo `Dictionary` y definiendo los métodos abstractos de la interfaz. Por ejemplo:

```
public class MiDiccionario extends Dictionary
{
   private Vector valor;
   private Vector clave;
   public MiDiccionario()  // constructor
   {
      valor = new Vector();
      clave = new Vector();
   }
   public int size()
   {
      return clave.size();  // número de elementos
   }
   public boolean isEmpy()
   {
      return clave.isEmpty();
   }
   public Object put(Object clave, Object valor)

   ...

}
```

17.9.2. Hashtable

Normalmente no es necesario definir una nueva clase diccionario, la clase histórica (Java 1.0) `Hashtable` extiende `Dictionary` y se puede utilizar para agrupar datos asociativos. `Hashtable` se comporta como una tabla *hash*; dispersa los elementos según el código que devuelve el método `hashCode()` del objeto `clave`.

Java 2 ha cambiado la declaración histórica de `Hashtable`, ahora implementa la interfaz `Map` y por consiguiente es compatible con las nuevas clases diccionario. La declaración:

```
public class Hashtable extends Dictionary implements Map, Cloneable,
                                                        Serializable
```

Nota de programación

La clase de los objetos `clave` debe disponer del método `int hashCode()`, utilizado para dispersar el elemento. También el método `equals()` para poder realizar búsquedas y comparaciones.

Constructores

El diseño de `Hashtable` se ha realizado con un factor de carga, por defecto, de `0.75`. El constructor de la clase permite establecer una capacidad inicial, y también establecer otro factor de carga (véase el Capítulo 12). Además, se puede inicializar un `Hashtable` con los elementos de un mapa.

`public Hashtable();`	tabla vacía, capacidad inicial 11, factor de carga 0.75.
`public Hashtable(int cap);`	tabla vacía, capacidad cap, factor de carga 0.75.
`public Hashtable(int cap, float factorCarga);`	tabla vacía.
`public Hashtable(Map mapa);`	crea una tabla con los elementos de mapa.

Métodos

Los métodos de `Hashtable` están especificados en la clase `Dictionary`, y en la interfaz `Map` a partir de Java 2. Además, el método `clone()` y los métodos para *serializar.* A continuación se hace mención a los más interesantes derivados de `Map`.

`public void putAll(Map mapa);`	inserta lo elementos de mapa.
`public boolean containsValue(Object valor);`	devuelve *true* si encuentra `valor` en el conjunto de valores de la tabla.
`public boolean containsKey(Object clave);`	devuelve *true* si encuentra `clave` en el conjunto de claves de la tabla.
`public Set keySet();`	devuelve un conjunto con las claves de la tabla.
`public Collection values();`	devuelve una colección con los valores de la tabla.
`public Set entrySet();`	devuelve un conjunto con los elementos de la tabla.

Además, redefine el método `toString()` para obtener una cadena con todos los pares de elementos de la tabla.

Ejemplo 17.12

Se crea un diccionario `HashTable` *cuyos elementos están formados por un campo valor de tipo* `String` *y una clave de tipo* `Double`*. Se realizan diversas operaciones con los métodos de* `Hashtable`.

Cada elemento representa a un ciclista, su nombre es el campo valor y el tiempo realizado en una crono es el campo clave. Los nombres y sus tiempos están en sendos *arrays;* para insertar en el diccionario se llama al método `put()`. Una vez creado, se realizan operaciones de búsqueda y eliminación de un elemento, obtener conjunto de claves, etc. Los elementos del diccionario se escriben en pantalla para mostrar los resultados de las operaciones.

```
import java.util.*;

public class DicHasTab
{
  public static void main(String[] args)
  {
```

```
    String [] ciclo = {"Lansen", "Messaria", "Ripolles", "Waritten",
                       "Delgado", "Kloster", "Bustaria", "Animador",
                       "Sanroma", "Juliani"};
    double [] tiempo = {45.40, 50.40, 47.0, 49.0, 51.20, 46.0, 48.0,
                         45.6, 52.30, 53.25};
    Hashtable tab = new Hashtable();
    // inserta los elementos en la tabla
    for (int i = 0; i < ciclo.length; i++)
      tab.put(new Double(tiempo[i]), ciclo[i]);
    System.out.println("Tabla hash creada: " + tab);
    // búsqueda por clave
    if (tab.containsKey(new Double(46.0)))
      System.out.println("Corredor encontrado: " +
                                    tab.get(new Double(46.0)));
    else
      System.out.println("Clave no está en la tabla");
    // elimina un elemento
    Object q = tab.remove(new Double(45.6));
    if (q != null) System.out.println("Elemento " +
                                              q + " eliminado");
    // Conjunto de claves
    Set cv;
    cv = tab.keySet();
    System.out.println("Conjunto de claves: " + cv);
    // Enumeración de valores
    Enumeration en;
    en = tab.elements();
    System.out.print("Ciclistas(valores): " );
    while (en.hasMoreElements())
    {
      System.out.print(en.nextElement());
      if (en.hasMoreElements()) System.out.print(", ");
    }
    System.out.println();
  }
}
```

Ejecución

```
Tabla hash creada: {53.25=Juliani, 46.0=Kloster, 50.4=Messaria,
47.0=Ripolles, 45.4=Lansen, 48.0=Bustaria, 52.3=Sanroma, 45.6=Animador,
49.0=Waritten, 51.2=Delgado}
Corredor encontrado: Kloster
Elemento Animador eliminado
Conjunto de claves: [53.25, 46.0, 50.4, 47.0, 45.4, 48.0, 52.3, 49.0, 51.2]
Ciclistas(valores): Juliani, Kloster, Messaria, Ripolles, Lansen,
Bustaria, Sanroma, Waritten, Delgado
```

17.9.3. Map

Los mapas o diccionarios almacenan información formada por parejas *(valor, clave)*. Java 2 enriquece los mapas creando una jerarquía exclusiva para ellos, la raíz es la interfaz `Map`, que, a diferencia de `Set` y `List`, no deriva de `Collection`. La Figura 17.5 muestra esta jerarquía.

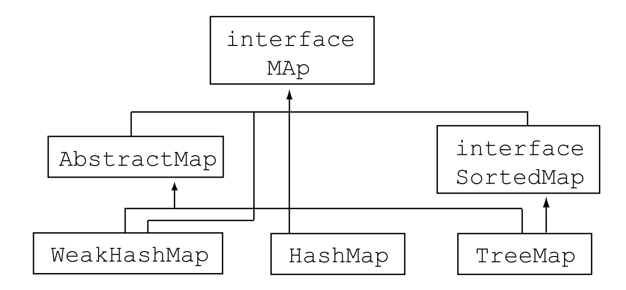


Figura 17.5 Jerarquía de *mapas*

La interfaz `Map` establece el comportamiento común de todos las implementaciones concretas de diccionarios, declara los siguientes métodos:

```
int size();
boolean isEmpty();
// métodos de búsqueda, por clave o por valor
boolean containsKey(Object clave);
boolean containsValue(Object valor);
Object get(Object clave);
// métodos que modifican el mapa, añaden o eliminan
Object put(Object clave, Object valor);
```
Si la clave existe reeplaza el valor antiguo y devuelve éste, en caso contraio `null`.
```
void putAll(Map mapa);
Object remove(Object clave);
```
Si la clave existe elimina el elemento y devuelve el valor asociado, en caso contraio `null`.
```
// métodos para obtener una "visión" del mapa, el conjunto
// de clave, o los valores, o bien el conjunto de elementos.
Set keySet();
Collection values();
Set entrySet();
```

AbstractMap

Esta clase abstracta implementa todos los métodos de la interfaz `Map`, excepto `entrySet()` que es necesario definirlo en las clases concretas. Además, define el método `toString()` de tal forma que devuelve una cadena con los pares de elementos que forman el mapa.

17.9.4. HashMap

Los mapas tipo `HashMap` organizan los elementos en una *tabla hash*, proporciona una eficiencia constante a las operaciones de búsqueda e inserción. Esta clase permite que haya claves y valores `null`, a diferencia de `Hashtable` que levanta una excepción si la clave es `null`. En general, el comportamiento de `HashMap` es similar a `Hashtable`. La declaración de la clase es la siguiente:

```
public class HashMap extends AbstractMap implements Map, Cloneable,
                                                    Serializable
```

Los elementos pares de un mapa, `(clave, valor)`, se guardan en un *array* de objetos de tipo `Entry`: `Entry [] tabla`; siendo esta una clase `static` declarada en `HashMap` que agrupa los

dos campos del elemento junto a su dirección *hash*. Además dispone de ciertos métodos que en ocasiones resultan útiles, los más interesantes son los siguientes:

`public Object getKey();`	devuelve la clave de la entrada (elemento).
`public Object getValue();`	devuelve el valor de la entrada (elemento).
`public Object setValue(Object nuevoVal);`	cambia el valor de la entrada por `nuevoVal` y devuelve el valor antiguo.
`public String toString();`	cadena con elemento, "clave = valor".

El acceso a los elementos del mapa sólo se puede realizar mediante un iterador al conjunto que devuelve el método `entrySet()`.

Constructores

El factor de carga de `HashMap` es, por defecto, `0.75`, aunque se puede poner otro en el constructor. La capacidad de la tabla se establece en el constructor, o bien se supone la capacidad por defecto si se utiliza el constructor vacío. También se puede crear un `HashMap` a partir de los elementos de otro *mapa*. Entonces los constructores son los siguientes:

`public HashMap();`	tabla vacía, factor de carga 0.75 y capacidad por defecto.
`public HashMap (int cap);`	tabla vacía de capacidad cap y factor de carga 0.75.
`public HashMap (int cap, float factorCarga);`	tabla vacía.
`public HashMap (Map mapa);`	crea una tabla con los elementos de mapa.

Métodos

La clase dispone de los métodos heredados de `AbstractMap` y los definidos al implementar los interfaz `Map`, `Cloneable` y `Serializable`. Por consiguiente, se puede buscar por clave o por valor con `containsKey()` y `containsValue()` respectivamente, insertar un elemento con `put()`, o bien insertar todos los elementos de otro mapa con `putAll()`. El método `remove()` elimina el elemento cuya clave es el argumento. Se puede obtener el conjunto de claves con `keySet()`, la colección de valores con `values()` y el conjunto de elementos con `entrySet()`. También, el método `get()` que obtiene el valor correspondiente a una clave.

La clase declara el método `static int hash(Object x)` para obtener una dirección *hash* adicional a `hasCode()`. El Ejemplo 17.13 crea un mapa `HashMap` con el que se realizan operaciones.

Ejemplo 17.13

Crear un mapa cuyos elementos son alumnos de una universidad; la clave de cada alumno es su número de matrícula.

Se simplifica el ejemplo de tal forma que un alumno es un `String` con su nombre y la clave es un número aleatorio de `1` a `99`. Los alumnos se encuentran en un *array* inicializado con nombres al azar. La entrada de elementos se realiza en un bucle de tantas iteraciones como número de nombres; al ser claves aleatorias, puede repetirse alguna de ellas. La redefinición de `toString()` permite escribir el mapa en pantalla en una sola acción; también se escribe elemento a elemento con un iterador del conjunto de claves. Se realiza la búsqueda de un elemento utilizando un iterador al conjunto de elementos; si se encuentra se cambia el valor (en este caso, el nombre) por otro nuevo.

```
import java.util.*;

public class MapaUniv
{
  public static void main(String[] args)
  {

    String [] univ = {"Ramiro", "Melendez", "Santos", "Armando",
     "Delgado", "Martina", "Bueno", "Alonso A", "Samuel", "Julian"};
    Map mp;
    mp = new HashMap(16);  // mapa vacío de capacidad inicial 16
    for (int i = 0; i < univ.length; i++)
    {
      int matricula;
      matricula = (int)(Math.random()*99 +1);
      if (! mp.containsKey(matricula))
        mp.put(new Integer(matricula), univ[i]);
      else i--;
    }
    System.out.println("Mapa creado: " + mp);
    // Iterador del conjunto de claves
    Set cv;
    cv = mp.keySet();
    Iterator it = cv.iterator();
    System.out.println("Recorre el Mapa con iterador de claves. ");
    while (it.hasNext())
    {
      Object clave;
      clave = it.next();
      System.out.print("(" + clave + "," + mp.get(clave) + ")");
      if (it.hasNext()) System.out.print(", ");
    }
    System.out.println();
    // cambio del valor de un elemento
    Set cel = mp.entrySet();
    Iterator ite = cel.iterator();
    System.out.println("Cambio de primera clave. ");
    Map.Entry elemento = (Map.Entry)ite.next();
    System.out.println("Valor elemento:" + elemento.getValue()
                            + ", clave: " + elemento.getKey());
    elemento.setValue("Zacarias");
    System.out.println("Mapa modificado: " + mp);
  }
}
```

Ejecución

```
Mapa creado: {15=Alonso A, 68=Bueno, 62=Armando, 19=Ramiro,
 31=Delgado, 95=Julian, 45=Martina, 97=Melendez, 70=Samuel, 88=Santos}
Recorre el Mapa con iterador de claves.
(15,Alonso A), (68,Bueno), (62,Armando), (19,Ramiro), (31,Delgado),
(95,Julian), (45,Martina), (97,Melendez), (70,Samuel), (88,Santos)
Cambio de primera clave.
Valor elemento:Alonso A, clave: 15
Mapa modificado: {15=Zacarias, 68=Bueno, 62=Armando, 19=Ramiro,
31=Delgado, 95=Julian, 45=Martina, 97=Melendez, 70=Samuel, 88=Santos}
```

17.9.5. TreeMap

Un mapa `TreeMap` mantiene en orden sus elementos para lo que utiliza la estructura *árbol rojinegro*. Este orden está determinado por el campo clave; puede ser en orden natural establecido por el método `compareTo()` del objeto `clave`, o bien por el comparador con el que se crea `TreeMap`. Es requisito imprescindible que la clave implemente a `Comparable`, o bien `Comparator`.

La interfaz `SortedMap` declara métodos que aprovechan la característica de ordenación de las claves de `TreeMap`, son los siguientes:

```
SortedMap subMap(Object desde, Object hasta);
SortedMap headMap(Object hasta);
SortedMap tailMap(Object desde);
Object firstKey();
Object lastKey();
```

Los tres primeros devuelven partes del *mapa*, es decir *submapas*, delimitados por la `clave`. El primer método devuelve el submapa formado por los elementos cuyas claves están en el rango `desde` (inclusive), `hasta` (exclusive). El segundo devuelve el submapa formado por elementos cuyas claves son menores que `hasta`. Y el tercero, por elementos cuyas claves son mayores o iguales que `desde`. Estos submapas son "visiones" del mapa de tal forma que cambios en el mapa se reflejan en el submapa y viceversa.

La declaración de `TreeMap` es como sigue:

```
public class TreeMap extends AbstractMap
                    implements SortedMap, Cloneable, Serializable
```

La definición de `TreeMap` incluye la definición de la clase `static Entry` que agrupa el par `(clave,valor)`. Esta clase dispone de métodos para obtener la clave `(getKey())`, para obtener el valor `(getValue())`, e incluso para cambiar el valor `(setValue())`. Son los mismos métodos de la clase homónima de `HashMap`; de hecho implementa la interfaz `Entry` declarado en `Map` que recoge esta funcionalidad común. Hay que tener en cuenta que la única forma de acceder a los elementos es mediante un iterador al conjunto de elementos que devuelve el método `entrySet()`.

Constructores

```
public TreeMap()
public TreeMap(Comparator c)
public TreeMap(Map mapa)
public TreeMap(SortedMap m)
```

En definitiva, un mapa puede crearse vacío, o bien con los elementos de otro mapa u otro mapa ordenado.

Métodos

`TreeMap` hereda los métodos de `AbstractMap` e implementa los métodos de los interfaces `SortedMap`, `Map`, `Cloneable` y `Serializable`. Las operaciones que implican búsquedas por clave: `containsKey()`, `get()`, `remove()` son muy eficientes, el tiempo de ejecución es *logarítmico* `O(log n)`. En general, la clase dispone de los mismos métodos que `HashMap`, y además los métodos descritos en `SortedMap`.

Ejemplo 17.14

Crear un mapa ordenado cuyos elementos son las cotas montañosas de una provincia y su respectiva altura en metros.

Los nombres de las montañas y sus cotas se encuentran en dos *arrays* paralelos. En esta ocasión el campo clave es el nombre de la montaña`(String);` por ello la ordenación de los elementos es alfabética, basada en el método `compareTo()` de la clase `String`. Se crea el mapa `TreeMap` vacío, a continuación se añaden los elementos con el método `put()`. El mapa se escribe directamente, esto es posible por estar redefinido `toString()`. Mediante llamadas a los métodos `subMap()`, `headMap()` y `trailMap()` se obtienen partes del mapa que se escriben en pantalla. También, se realizan búsquedas y eliminación de elementos.

```
import java.util.*;

public class MapaOrdenado
{
  public static void main(String[] args)
  {
    String [] montan =
        {"Teleno", "Melarita", "Ocejon", "Peñahueca", "Almanara",
         "Ocenillo", "Bustarano", "Reinosa", "Urbieta", "Galarita"};
    int [] alto =
       {1789, 1235, 1790, 2211, 2200, 1780, 1450, 2507, 1478, 2010};
    TreeMap mapa;

    mapa = new TreeMap();  // mapa vacío
    for (int i = 0; i < montan.length; i++)
         mapa.put(montan[i], new Integer(alto[i]));
    System.out.println("\t Mapa creado \n" + mapa);

    SortedMap su1, su2, su3;
    su1 = mapa.subMap("B", "R");    // desde 'B' hasta 'R' (exclusive)
    System.out.println("\t Submapa en el rango [B ... R) \n” + su1);
    su2 = mapa.headMap("R");        // claves menores que 'R'
    System.out.println("\t Submapa de claves menores que R \n" + su2);
    su3 = mapa.tailMap("R");        // claves mayores o iguales que 'R'
    System.out.println("\t Submapa de claves mayores que R \n" + su3);

    System.out.println("Borra primer elemento: " +
    mapa.remove(mapa.firstKey()));  // borra el primer elemento
    System.out.println("\t Mapa actual \n" + mapa);
  }
}
```

Ejecución

```
         Mapa creado
{Almanara=2200, Bustarano=1450, Galarita=2010, Melarita=1235, Ocejon=1790,
Ocenillo=1780, Peñahueca=2211, Reinosa=2507, Teleno=1789, Urbieta=1478}
          Submapa en el rango [B ... R)
{Bustarano=1450, Galarita=2010, Melarita=1235, Ocejon=1790, Ocenillo=1780,
Peñahueca=2211}
          Submapa de claves menores que R
{Almanara=2200, Bustarano=1450, Galarita=2010, Melarita=1235, Ocejon=1790,
```

```
Ocenillo=1780, Peñahueca=2211}
         Submapa de claves mayores que R
{Reinosa=2507, Teleno=1789, Urbieta=1478}
Borra primer elemento: 2200
         Mapa actual
Bustarano=1450, Galarita=2010, Melarita=1235, Ocejon=1790, Ocenillo=1780,
Peñahueca=2211, Reinosa=2507, Teleno=1789, Urbieta=1478}
```

17.10. COLECCIONES PARAMETRIZADAS

Las colecciones históricas y las desarrolladas en Java 2 guardan, internamente, los elementos en *arrays* o vectores de tipo `Object`. Con esto se consigue la máxima generalización ya que `Object` es la clase base de cualquier objeto Java y se permite que una colección guarde objetos de cualquier tipo, por ejemplo una lista puede contener cadenas (`String`), números racionales ... :

```
LinkedList lis = new LinkedList();
Racional r = new Racional(3,7);

lis.addElement(r);
lis.addElement(newString("Mi globo");
```

Al recuperar elementos de la lista es necesario discernir el tipo concreto de elemento, por ejemplo:

```
Object q;
Racional t;

q = lis.getFirst();
if (q instanceof Racional)
   t = (Racional) q;
...
```

En muchas aplicaciones de colecciones los elementos son del mismo tipo, un conjunto de enteros, una lista de palabras..., a pesar de lo cual la recuperación de elementos siempre necesita una conversión de `Object` a ese tipo. Por esta razón, y otras como que al compilar pueda realizar comprobaciones de tipo, Java 1.5 amplía la declaración de todas las clases e interfaces relacionados con colecciones para dotarlas de la posibilidad de parametrizar el tipo que va a tener los elementos de una colección. Por ejemplo, un vector que vaya a guardar elementos de tipo `Complex` se declarará e instanciará:

```
Vector<Complex> zz;
zz = new Vector<Complex>(19);
```

La funcionalidad de `Vector` no cambia, eso sí el compilador comprobará tipos de datos al realizar operaciones, como por ejemplo añadir elementos:

```
Complex nz = new Complex(1,-1);
zz.addElement(new Integer(8));  //error de compilación, tipo mismatch
zz.addElement (nz);             // correcto
```

La recuperación de elementos ya no necesita conversión de `Object` a `Complex`:

```
nz = zz.elementAt(0);
```

17.10.1. Declaración de un tipo parametrizado

Al nombre de la colección le sigue el tipo de los elementos entre paréntesis angulares (< tipo>):

```
Coleccion<tipo> v;
```

Si se parametrizan dos tipos, como ocurre con los mapas, se separan con coma:

```
Coleccion<tipoClave,tipoValor> cc;
```

Por ejemplo:

```
Stack<Double> pila;
SortedMap<Integer,String> mapa;
```

Realmente con el tipo parametrizado es como si se hubiera declarado otra clase, en consecuencia las instancias de colecciones parametrizadas se crean con esa clase, es decir new Coleccion<tipo> crea la instancia. Por ejemplo:

```
pila = newStack<Double>();
mapa = newTreeMap<Integer,String>();
```

Sintaxis

A continuación del nombre de la clase se especifican los tipos parametrizados entre paréntesis angulares:

```
Coleccion<tipo1...> var;
```

Para crear instancias, new *Coleccion*<tipo1...>();

Por ejemplo:

```
Set<Integer> cn = new TreeSet<Integer>();
```

Ejemplo 17.15

Declarar los mapas de los ejemplos 17.13 y 17.14 especificando el tipo del campo clave y del campo valor.

Los tipos de la clave son Integer y String respectivamente. Los tipos del campo valor son String y Integer respectivamente. Entonces, las declaraciones son las siguientes:

```
Map<Integer,String> mp;
mp = new HashMap<Integer,String>(16);

TreeMap<String,Integer> mapa;
mapa = new TreeMap<String,Integer>();
```

RESUMEN

Una colección agrupa objetos relacionados que forman una única entidad; por ejemplo un array de objetos, un conjunto de números complejos, etc. Las colecciones incluyen clases contenedoras, iteradores para acceder a los objetos en el interior de los contenedores y algoritmos para manipular los objetos (métodos de clases). Las clases *Colección* guardan objetos de cualquier tipo; de hecho el elemento base es `Object` y por consiguiente se podrá añadir a la colección un objeto de cualquier tipo. Las colecciones históricas (Java 1) mas importantes son `Vector`, `Stack`, `Dictionary`, `HashTable` y la interfaz `Enumeration` para recorrer los elementos de una colección.

Java 2 incorpora nuevas clases colección, diseñadas sin el comportamiento de sincronización. Hay tres tipos generales de colecciones: *conjuntos*, *listas* y *mapas*; los interfaces `Set`, `List` y `Map` describen las características generales de éstos. Además, la interfaz `Collection` especifica el comportamiento común de las colecciones.

Las clases `Arrays` y `Collections` agrupan algoritmos útiles que se aplican, respectivamente, a arrays de los tipos primitivos y a todo tipo de colecciones. Por ejemplo:

```
Arrays.sort(w);
Collections.reverse(lista2);
```

Los elementos que se comparan deben implementar la interfaz `Comparable`, que declara el método `compareTo()`, o bien la interfaz `Comparator` que declara el método `compare()`.

La clase `Vector` se comporta como un array de elementos genéricos que se redimensiona automáticamente. `Stack` representa el tipo abstracto *Pila*, y deriva de `Vector`. Ambas son colecciones históricas que se han redefinido para encuadrarlas en el conjuntos de colecciones de Java 2.

La llamada al método `elements()` de una clase colección devuelve un `Enumeration` para recorrer o acceder a sus elemento sin modificarlos. Los métodos de `Enumeration` son los siguientes:

```
boolean hasMoreElements();
Object nextElement();
```

`Iterator` es otro iterador, definido a partir de Java 2, que permite acceder a los elementos y también eliminarlos. Los métodos que declara son los siguientes:

```
boolean hasNext();
Object next();
void remove();
```

La interfaz `Collection` es la raíz de la jerarquía de colecciones y describe el comportamiento común de las colecciones. Hay tres tipos de colecciones descritas por los interfaces `List`, `Set` y `Map`; las dos primeras derivan de `Collection`.

Las listas son de dos tipos: secuenciales `(ArrayList)` y enlazadas `(LinkedList)`. El concepto general de lista está representado por la interfaz `List`. El iterador `ListIterator` está diseñado par recorrer cualquier lista, en particular una lista enlazada. Con este iterador se puede avanzar o retroceder por los elementos.

La interfaz `Set` declara las operaciones generales de los conjuntos. Todas las operaciones matemáticas de los conjuntos: *unión*, *intersección*... se realizan con los métodos de las clases concretas `HashSet` y `TreeSet`. La clase `HashSet` guarda los elementos de un conjunto sin mantener un orden. Los conjuntos de tipo `TreeSet` se diferencian de los `HashSet` por mantener en orden los elementos. La ordenación puede ser ascendente, es decir, en el orden

natural determinado por la interfaz `Comparable`; o bien el orden que establece la implementación de la interfaz `Comparator`.

Las colecciones históricas definen los diccionarios con la clase abstracta `Dictionary` y la clase concreta `Hashtable`. Java 2 declara la interfaz `Map` con el fin de especificar los métodos comunes a los mapas. Los mapas tipo `HashMap` organizan los elementos en una *tabla hash*, proporciona una eficiencia constante a las operaciones de búsqueda e inserción. Esta clase permite que haya claves y valores `null`. Un mapa `TreeMap` mantiene en orden a sus elementos para lo que utiliza la estructura *árbol roji-negro*. La interfaz `SortedMap` declara métodos que aprovechan la característica de ordenación de las claves de `TreeMap`, son los siguientes:

```
SortedMap subMap(Object desde, Object hasta);
SortedMap headMap(Object hasta);
SortedMap tailMap(Object desde);
Object firstKey();
Object lastKey();
```

Java 1.5 amplía la declaración de todas las clases e interfaces relacionados con colecciones para dotarlas de la posibilidad de parametrizar el tipo de los elementos de una colección. Por ejemplo, un vector que vaya a guardar elementos de tipo `Complex` se declarará e instanciará:

```
Vector<Complex> zz;
zz = new Vector<Complex>(19);
```

Al nombre de la colección le sigue el tipo de los elementos entre paréntesis angulares (`< tipo>`). Los tipos se separan por coma (`,`):

```
Coleccion<tipo> v;
Coleccion<tipo1,tipo2> w;
```

EJERCICIOS

17.1. Declarar la clase `Pagina` y un array de tipo `Pagina` de `n` elementos, siendo `n` un dato de entrada. A continuación, asignar a cada elemento del array un objeto `Pagina`.

17.2. Modificar la declaración de la clase `Pagina` de tal forma que implemente la interfaz `Comparable`. Crear un array de tipo `Pagina` y llamar al método `sort()` para que esté ordenado.

17.3. Declarar un array de tipo `String` de 40 elementos. Llamar al método `fill()` para inicializar la primera mitad a "`Domingo`" y la segunda a "`Lunes`".

17.4. Realizar la declaración de una lista enlazada cuyos elementos sean de tipo `Persona`. Añadir elementos a lista y después declararla de sólo lectura.

17.5. ¿Es posible añadir alternativamente, a una colección `ArrayList`, elementos de tipo `Integer` y de tipo `Double`?

17.6. ¿Qué diferencias existen entre un iterador de tipo `Enumeration` y otro de tipo `Iterator`?

17.7. ¿Qué diferencias existen entre un iterador de tipo `Iterator` y otro de tipo `ListIterator`?

17.8. ¿Por qué las colecciones diseñadas en Java 2 no están sincronizadas?

17.9. Declare un conjunto cuyos elementos no estén ordenados. ¿Cómo proceder para que dicho conjunto esté sincronizado?

17.10. Señalar las ventajas de utilizar tipos parametrizados en las colecciones. ¿Tiene algún inconveniente?

17.11. Declarar variables mapa cuyos elementos son de tipo `(Double, String)` y `(Integer, Racional)` respectivamente; la declaración se ha de realizar con tipos parametrizados.

PROBLEMAS

17.1. Se desea almacenar en un *mapa* los atletas participantes en un cross popular. Los datos de cada atleta: *Nombre, Apellido, Edad, Sexo, Fecha de nacimiento* y *Categoría (Junior, Promesa, Senior, Veterano)*. La clave es el *Apellido* del atleta. Realizar las declaraciones necesarias y una aplicación que cree el mapa y posteriormente lo muestre por pantalla.

17.2. Las palabras de un archivo de texto se quieren mantener en una tabla *hash* para poder hacer consultas rápidas. Los elementos de la tabla son la cadena con la palabra y el número de línea en el que aparece. Escribir una aplicación que lea el archivo, según se capture una palabra y su número de línea se insertará en la tabla dispersa. Considerar que una palabra se separa de otra por un espacio en blanco.

17.3. Escribir una aplicación para realizar las operaciones del tipo de dato conjunto: *unión, intersección, diferencia, pertenencia* de un elemento e inclusión. Los elementos del conjunto serán números enteros aleatorios del 1 al 999.

17.4. Crear un *mapa ordenado* cuyos elementos sean títulos de libros y su respectivo autor. Considerar el nombre del autor como campo clave. La aplicación dará entrada a los elementos del *mapa* y realizará operaciones que obtengan *submapas* por rango, o a partir de una clave dada.

17.5. La clase `Racional` representa un número racional, y tiene dos atributos: numerador y denominador. Declarar la clase `Racional` y la clase `Compara` para definir el método `compare()` de la interfaz `Comparator` aplicado a dos números racionales (realmente a dos objetos `Racional`). Escribir una aplicación que cree dos conjuntos ordenados de tipo `Racional`, utilizando como comparador `Compara`, realice operaciones *unión, intersección y diferencia*. La aplicación deberá, también, obtener subconjuntos que mostrará en pantalla mediante un iterador.

BIBLIOGRAFÍA

AHO, HOPCROFT, ULLMAN (1998). *Estructuras de datos y Algoritmos*. Addison-Wesley.

BAILEY, Duanes (1999). *Data Structures in Java*. McGraw-Hill.

BARNES, David (2000). *Object-Oriented Programming in Java: An introduction.* Prentice-Hall.

BUDD, Timoty (2002). *Object-Oriented Programming.* Third Edition. Addison-Wesley.

DEITEL, Harvey (2002). *JAVA. How to program.* Fourth edition, Prentice-Hall.

GILBERT, Stephen y MCCARTY, Bill (1997). *Object-Oriented Programming in Java.* Waite Group Press.

GILBERT, Stephen y MCCARTY, Bill (1998). *Object-Oriented Design in Java.* Waite Group Press.

HEILEMAN, G. (1998). *Estructuras de datos, algoritmos y programación orientada a objetos*. McGraw-Hill,

HORSTMANN, Cornell (2006). *Core Java 2 Volumen I–Fundamentos*. Pearson Educación.

HUBBARD, John R. (2001). *Data Structures with Java*. McGraw-Hill, Schaum´s OuTlines.

JOYANES, Luis (2003). *Fundamentos de programación*. 3ª edición; McGraw-Hill.

JOYANES, Luis y FERNÁNDEZ, Matilde (2001). *Java 2, Manual de programación*. McGraw-Hill.

JOYANES, Luis; FERNÁNDEZ, Matilde; SÁNCHEZ, Lucas y ZAHONERO, Ignacio (2005). *Estructuras de datos en C.* McGraw-Hill. Colección *Schaum*.

JOYANES Luis y ZAHONERO Ignacio (2002). *Programación en Java2* . McGraw-Hill.

JOYANES Luis y ZAHONERO Ignacio (2004). *Algoritmos y estructuras de datos, una perspectiva en C*. McGraw-Hill,

KUTH, Donald (1985). *Algoritmos fundamentales volumen I*. Editorial Reverté.

LAFORE, Robert (2003). *Data Structures & Algorithms in Java*. Second Edition. Sams.

LIPSCHUTZ, S (1989). *Estructuras de datos*. McGraw-Hill.

WEISS, M.A. (1995). *Data Structures and Algorithm Análisis*. Benjamín/Cummings.

WEISS, M.A (1999). *Data Structures & Algorithm Analysis in Java.* Addison-Wesley.

WEISS, M.A (2000). *Estructuras de datos en Java*. Addison-Wesley.

ZUKOWSKI, John (1999). *Programación en Java 2*. Anaya Multimedia.



Índice de términos